Naval Ocean Systems Center
San Diego, CA 92152-5000

(4)

DTIC FILE COPY

Technical Document 1837
June 1990

Advanced Numerical Techniques of Performance Evaluation

Volume I

AD-A230 292

University of Washington

DTIC
S ELECTE D
DEC 28 1990
D

Approved for public release; distribution is unlimited.

The views and conclusions contained in this report are those of the contractors and should not be interpreted as representing the official policies, either expressed or implied, of the Naval Ocean Systems Center or the U.S. Government.

90 12 26 024

NAVAL OCEAN SYSTEMS CENTER

San Diego, California 92152-5000

J. D. FONTANA, CAPT, USN
Commander

R. M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

Contract N66001-87-D-0136 was carried out by University of Washington, Department of Computer Sciences, Seattle, WA 98195, under the technical coordination of T. Sterrett, Computer Systems Software and Technology Branch, Code 411, Naval Ocean Systems Center, San Diego, CA 92152-5000.

Released by
R. A. Wasilausky, Head
Computer Systems Software
and Technology Branch

Under authority of
A. G. Justice, Head
Information Processing and
Displaying Division

FS

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 18(8), 713-732 (AUGUST 1988)

PRESTO: A System for Object-oriented Parallel Programming

BRIAN N. BERSHAD, EDWARD D. LAZOWSKA AND HENRY M. LEVY
Department of Computer Science, FR-35, University of Washington, Seattle WA 98195, U.S.A.

SUMMARY

PRESTO is a programming system for writing object-oriented parallel programs in a multiprocessor environment. PRESTO provides the programmer with a set of pre-defined object types that simplify the construction of parallel programs. Examples of PRESTO objects are threads, which provide fine-grained control over a program's execution, and synchronization objects, which allow simultaneously executing threads to co-ordinate their activities.

The goals of PRESTO are to provide a programming environment that makes it easy to express concurrent algorithms, to do so efficiently, and to do so in a manner that invites extensions and modifications. The first two goals, which are the focus of this paper, allow a programmer to use parallelism in a way that is naturally suited to the problem at hand, rather than being constrained by the limitations of a particular underlying kernel or hardware architecture. The third goal is touched upon but not emphasized in this paper.

PRESTO is written in C++; it currently runs on the Sequent shared-memory multiprocessor on top of the Dynix operating system. In this paper we describe the system model, its applicability to parallel programming, experiences with the initial implementation, and some early performance measurements.

KEY WORDS Design Languages Measurement Performance Parallel computing Software parallelism Speed-up Efficiency

INTRODUCTION

PRESTO is a programming system for writing object-oriented parallel programs in a multiprocessor environment. PRESTO consists of an object-oriented language (C++[1]), a library of basic tools constructed in this language, a run-time system providing efficient support, and, most important but least tangible, a programming methodology.

Our first goal in designing and implementing PRESTO was to apply our experiences in building distributed object-oriented systems[2, 3] to the world of multiprocessors. In distributed systems, an object-oriented programming paradigm makes it easier to think about and to express concurrent algorithms. Problem decomposition and run-time synchronization details can be neatly described by an object model. Each object is responsible for solving some small part of an overall problem, and each is responsible for maintaining its own (and only its own) internal consistency. These are exactly the qualities that are needed when building parallel applications for a multiprocessor.

0038–0644/88/080713–20$10.00
© 1988 by John Wiley & Sons, Ltd.

Received 16 October 1987
Revised 4 February 1988

DTIC COPY INSPECTED 6

Availability Codes
Dist
Avail and/or Special
A-1

Our second goal was to provide efficient concurrency and synchronization mechanisms. The primitives provided by many existing parallel programming systems are so expensive that they become the major factor in determining the structure of applications. (For example, one may be forced to design algorithms that use only as many threads of control as there are physical processors.) Our experience (and common sense) suggested that the construction of parallel applications — a daunting task under the best of circumstances — was made considerably more difficult by such constraints. PRESTO allows the programmer to use parallelism in the manner most natural to the problem at hand, with minimal external performance constraints arising from an underlying kernel or hardware architecture.

Our third goal was to provide an 'open' environment that could be used as a 'toolkit' for building efficient support for a variety of 'models' of parallel programming. Most parallel programming systems present themselves in terms of a fixed set of primitives running on top of a closed run-time kernel. The primitives together with the kernel define a 'model' of parallel programming that, while pleasing to the implementor, may not always be satisfactory to the application programmer. In PRESTO, it is possible to redefine the behaviour of the lowest level system primitives. New constructs (and thus new abstractions) for constructing parallel applications can be introduced quickly, and without the level of overhead normally associated with a layered system.

This paper concentrates on describing PRESTO in terms of the first two goals: its object-orientation and its performance, and the impact of these characteristics on parallel programming. The third goal, that of providing an open environment for building parallel programming systems, is fully described in a companion paper.[4] The examples in this paper are drawn from what could be called the 'default' PRESTO programming model — a Mesa-like environment where threads, monitors and condition variables define the basic primitives for the programmer.

The remainder of this paper discusses the implementation language for PRESTO (C++), the use of objects in building parallel programs, a few of the default PRESTO objects, the use of PRESTO on a multiprocessor and some preliminary performance measurements.

PRESTO AND THE C++ PROGRAMMING LANGUAGE

PRESTO is implemented in C++. To quote from the C++ reference manual:

> C++ is a superset of the C programming language that retains the efficiency and notational convenience of C, while providing facilities for type checking, data abstraction, operator overloading and object-oriented programming.

We chose C++ for three reasons, each touched upon in the above quotation. First, we wanted an object-oriented programming language. Secondly, C++ is implemented as a preprocessor to C, making it portable to any system with a C compiler. (Although PRESTO exists now on only one machine, we intend to port it to other multiprocessors as they become available to us.) Thirdly, we wanted PRESTO to be widely used, and C++ is relatively easy to learn for C programmers.

Object-oriented programming in C++ is made possible by the concept of a *class*. A class is a user-defined data type allowing the programmer to specify an object in terms of its data representation and operations. As an example, the class definition for a simple stack of integers might appear as

```
// This is a comment.
class Stack     {
        //Private Data
        int                 st_size;                 // maximum size
        int                 st_sp;                   // current stack pointer
        int                 *st_elements;            // data
        // Private Operations
        void                st_growstack();          // make the stack larger
public:
        // Public Operations
        Stack (int maxsize)                                // CONSTRUCTOR
           { st_size = maxsize;
             st_sp = 0;
             st_elements = new int[st_size];
         ~Stack()                                          // DESTRUCTOR
           { delete st_elements;}
        int              depth()                           // current depth
           { return st_sp; }
        virtual void     (push(int newelement)
           { if (st_sp == st_size)
                         st_growstack();                   //  ensure  we  have
                                                           // room
             st_elements[st_sp++] = newelement;
           }
        virtual int         pop()
           { if (depth() == 0)                       // nothing left?
                         // ERROR HANDLER HERE
             else
                            return st_elements[--st_sp];
           }
};
```

Each instance of a Stack has its own private version of the class's variables. An object's operations are shared by all instances of the object's class. The declarations in the private section specify those parts of the class accessible only from within the operations themselves. If another object creates a stack,

```
Stack *S = new Stack(100);     // Invoke the CONSTRUCTOR; maxsize = 100
```

then only the operations

```
S->push(x);
x = S->pop();
sz = S->depth();
delete S;              // Invoke the DESTRUCTOR
```

can be performed on S. S's private operations and private data can be referenced from within these public operations. The process of performing an operation is called an *invocation*.

The keyword virtual in the class definition indicates that push() and pop() can be redefined by any classes that are defined as a sub-class of Stack. In C++, a sub-class can be derived from a super-class, allowing classes to exist hierarchically. The sub-class's qualities are those it defines for itself, as well as those inherited from its super-class. If a sub-class redefines any of its inherited virtual operations, the new definitions take precedence. The sub-class satisfies the isa relation on its super-class. That is, if class SynchronizedStack is a sub-class of class Stack,

```
// Sub-class                    isa        Super-class
class SynchronizedStack          :         Stack {
        //
        // Serialize access to the stack
        //
};
```

then an instance of class SynchronizedStack can be used anywhere an instance of class Stack is expected — SynchronizedStack isa Stack. A synchronized stack object can ensure that concurrent operations on the private data defined in its super-class are serialized, without having to redefine the implementation of those operations. A more complete definition for SynchronizedStack appears later in this paper.

An object's operations may also include *constructor* and *destructor* routines. These are procedures that are called automatically when the object is created and deleted, respectively, allowing the object to specify an initialization and termination sequence. New objects can be created on the call-stack by declaring them within a basic block, statically by declaring them outside of any block, or on the heap, by using the new operator.

The definition for an operation can either be included in the class definition itself or given elsewhere. To define the operation st_growstack() separately from the declaration, it is necessary to use the qualifying syntax '::'.

```
//
//    Class::Operation qualifies Operation under Class
//
void                              // return type
Stack::st_growstack()             // Class::Operation
{
        int j;
        int newsize = st_size * 2;               // double size
        int *newstack = new int[newsize];
        for (j = 0; j < st_size; j++)            // copy old stack
                newstack[j] = st_stack[j];       // into new
        st_size = newsize;
        delete st_stack;
        st_stack = newstack;
}
```

(These programming segments are meant only to provide enough exposure to C++ so that the reader can understand the remaining examples in this paper. For complete information, see Reference 1.)

C++ is an inherently sequential language. Unlike languages such as Emerald or Modula2+,[5] C++ has no notion of concurrency or synchronization.

It would have been possible to extend the language in this direction by modifying the compiler, but we felt that the language was sufficiently rich that our objectives could be achieved without changing it. Furthermore, including knowledge about concurrency and synchronization in the compiler would have seriously limited future extensions to PRESTO itself (unless one were willing to again modify the compiler).

Available as part of the Standard C++ distribution is a set of classes for defining concurrent objects on a uniprocessor. These objects execute as co-routines, and they are limited in terms of how they can be used. (For example, objects can only be single-threaded and synchronize only with messages.) Making these objects work on a multiprocessor would have been possible (and indeed has been done elsewhere[6]) but would have precluded us from realizing the goals of efficient primitives implemented on an open system. We have used PRESTO to define classes that efficiently mimic the behaviour of those provided as part of the C++ distribution, without assuming that this behaviour is the 'PRESTO-definitive' mode of parallel programming.

Since PRESTO is written in C++, it is most naturally used with applications written in that language. Although it is possible to use the system from other languages (such as C or Pascal), many of PRESTO's concepts will be difficult and time-consuming to express. For this reason, users are encouraged either to write completely in C++, or to build application-specific interfaces between languages.

EXPLOITING THE OBJECT MODEL IN PARALLEL PROGRAMS

PRESTO provides the programmer with several classes useful for writing parallel programs. These classes, and the environment in which they execute, help support two of the major goals of PRESTO — efficient execution and comfortable abstractions for expressing concurrency.

In PRESTO, all objects execute in a single address space shared by all processors, allowing for fast inter-object communication and synchronization through shared storage. The object model allows objects to exist in a 'safe' environment, making it difficult (although not impossible) for objects to trounce one another haphazardly. In a sequential object-oriented system, an object hides its data and its implementation. In PRESTO, an object hides not only its data and its implementation, but also its *execution*. That is, when a caller invokes an operation on an object, the caller is unaware whether that operation executes sequentially or in parallel. The implementor of an object determines the extent of parallelism that is appropriate to the object, much as he/she decides what data structures best suit the needs of the object. Dealing with concurrency in this manner simplifies the task of writing parallel programs.

The following sections describe the major classes used by PRESTO programs, and discuss how their design addresses the goals of the system.

The thread class

Thread objects (threads) are the building blocks of PRESTO parallel programs. As the basic unit of execution, threads consist conceptually of a program counter and a stack of invocation records. There are two essential operations that can be performed on a thread. A thread can be *created*, allowing the creator to specify the thread's qualities, such as its name and maximum storage requirements. Once created, a thread can be *started* executing some operation of some object, wherein it executes in parallel with the starting thread. Start, in fact, is an operation defined for threads; parameters include the object, the operation where the thread is to be started, and any parameters expected by that operation. For example,

```
Stack •S = new Stack(100);
// Create a new thread named "Pusher" having id TID.
Thread •t = new Thread("Pusher", TID);

// Let t be responsible for pushing 43 onto the stack.
t->start(S, S->push, 43);
```

As noted earlier, PRESTO extends conventional object-oriented programming by allowing an object to hide and control not only its data and its implementation, but also its execution. The user of an object chooses between synchronous and asynchronous invocations, and the implementor of an object chooses between sequential and parallel execution. Table I shows how these choices fit together, and how their combinations affect the overall execution of a program.

An object cannot tell whether it is being invoked synchronously or asynchronously, and the user of an object cannot tell whether an invocation is being performed sequentially or in parallel. For example, the user of a matrix object is probably not concerned with whether the object implements multiplication by using hundreds of parallel threads or a single thread executing over the whole problem. Only the ultimate product is important. It is the responsibility of the matrix object to determine where, when and how much parallelism is dictated by a given invocation of the multiply operation. In cases such as this, synchronous invocation with parallel execution is most appropriate. The following code segment shows how a parallel matrix multiplication operation might be defined in PRESTO:

Table I. Control over execution

User of object (*U*) chooses	Implementor of object (*I*) chooses	Effect
synchronous	sequential	*U* invokes *I*'s operation. *U* blocks until *I* finishes, *I* runs single-threaded with *U*'s thread.
synchronous	parallel	As above, only *I* creates multiple threads and starts them executing its own operations. These threads compute in parallel.
asynchronous	sequential	*U* creates a new thread and starts it executing *I*'s operation. *U* continues, and *I* runs with the single thread that *U* started within it. When *I* returns, its thread is destroyed.
asynchronous	parallel	*U* creates a new thread and starts it executing *I*'s operation. *U* continues, while *I* creates multiple threads and starts them executing *I*'s operations.

```
class Matrix    {
        int     **ma_elems;
        int     ma_rows;
        int     ma_cols;
        void    ma_dotproduct(int *el, Matrix *M, int i, int j)
                {    // Compute the dot product of the i'th row
                     // of "this" (in ma_elems) and
                     // the j'th column of M. Store in *el.
                }
public:
        Matrix(int rows, int columns);                // CONSTRUCT new matrix
        int     numcolumns();
        int     numrows();
        Matrix *multiply(Matrix *M);                  // multiple "this" by M
        // other operations. . .
}
// Create a separate thread to compute each element in the
// product matrix in parallel.
Matrix*
Matrix::multiply (Matrix *M)
{
        Matrix *P = new Matrix(ma_rows, M->numcolumns());     // product
        for (int i = 0; i< ma_rows); i++)
            for (int j = 0; j < M->numcolumns()     {
                Thread *t = new Thread("multiplier");
                t->start(this, this->ma_dotproduct,
                    &(P[i][j]), M, i, j); // arguments to operation
//
//Wait here until all threads terminate
//
return P;
      }
```

Alternatively, the insertion of a new entry into a directory object is a non-parallel operation, best handled asynchronously by the object doing the insertion, not by the directory object itself. The insert invocation might appear as:

```
// Create the directory
Directory *dir = new Directory("my_directory");

// Create a thread to do the insertion
Thread *t = new Thread("dir_inserter");

// Start the thread doing the insertion of some file.
t->start(dir, dir->insert, someFileName, someFileContents);

// Run in parallel with the dir->insert routine.
// Its termination is transparent.
```

A thread may only be started once. It executes either until it terminates itself, or until it returns from the operation in which it was started. A thread may join on another thread, causing the joining thread to block until the joinee finishes. The joinee's return value from the operation in which it started is returned to the joining thread when it resumes.

Each thread has its own call-stack. Any objects declared on that call-stack are visible *only* from within the thread to which the call-stack belongs. Objects requiring references from more than one thread must be heap allocated or static. If data is to be shared between threads, then it should be declared as such within the object's definition.

The synchronization class

Although a thread can be executing in only one object at a time, it is possible to have multiple threads executing within a single object simultaneously. In a multiprocessor system, true concurrency can occur. To provide a controlled environment for multi-threaded objects, PRESTO provides two basic classes of synchronization objects: relinquishing and non-relinquishing locks.

Relinquishing locks

A thread executes until it is pre-empted, terminates or voluntarily relinquishes the processor by performing an operation on a relinquishing object. The simplest relinquishing objects are those defined by the class Lock. The two primary operations on locks are lock() and unlock().

```
// l is a reference to a Lock (Lock* l)
l->lock();
        // critical code
l->unlock();
```

Return from a lock() operation indicates that the caller *holds* the lock. A lock may be held by only one thread at a time. A thread trying to lock an already held lock relinquishes the processor on which it is running, allowing another ready thread to execute on that processor. The relinquishing thread is made ready for execution when the lock becomes free.

Non-relinquishing locks

Hardware atomic locks serve as the basis for the non-relinquishing synchronization object Spinlock. Spinlocks have a potential performance advantage over simple relinquishing locks. It is less expensive to acquire and release a non-relinquishing lock. Further, if the average waiting time is less than the time to relinquish and reacquire a processor, non-relinquishing locks are more efficient.

As with simple relinquishing locks, there are two operations on spinlocks, lock() and unlock().

```
// s is a reference to a Spinlock (Spinlock *s)
s->lock ();                          // spin here if already locked
```

```
        // critical code
    s->unlock();
```

The thread that most recently locked the spinlock is the lock's owner. A thread relinquishes ownership by unlocking the spinlock. A spinlock can have only one owner, but unlike relinquishing locks, a thread trying to lock an owned spinlock consumes CPU cycles polling the lock until it becomes free. If a thread tries to lock a spinlock that it already owns, the thread will spin forever. The implementation of spinlocks causes a thread to become non-pre-emptible once it acquires one.

More sophisticated synchronization classes

Monitors and condition variables

Although straightforward and easy to understand, simple relinquishing locks can be difficult to use and thus prone to misuse. A more refined relinquishing synchronization mechanism is available through monitors and condition variables.* These work together to provide Mesa-like synchronization semantics.[7–9] As noted in the introduction, we view this as the 'default' PRESTO programming model, but we encourage users to build (and share) support in PRESTO for other parallel programming models when this is dictated by their applications.

In PRESTO, a section of critical code is surrounded by an entry and exit invocation on a monitor object. If several operations must coexist within the same monitor, the programmer is obligated explicitly to name the monitor within each operation. Although there is no direct compile-time support for monitors, it is not necessary for the programmer to make explicit calls to a monitor's entry and exit routines when writing a monitored object. PRESTO provides a type MONITOR that can be used to guard access to blocks of code:

```
{ // example of monitored block of code controlled by Monitor *m;
  MONITOR ENTRY(m);       // ENTRY is a nice sounding dummy variable
      // . . code here
}     // m is automatically released when ENTRY goes out of scope here
```

is equivalent to

```
{
  m->entry();
      // . . code here
  m->exit();
}
```

but is syntactically cleaner. Furthermore, the first form makes it impossible that the programmer will forget to explicitly release the monitor, *even if* the code returns from within the block. The constructor for a MONITOR object enters the named monitor, and remembers it. When the MONITOR object goes out of scope (at the bottom of a block), its destructor is automatically called. Within the destructor, the entered monitor is exited.

---

* Condition variables are really condition objects, but the former terminology is well-established, and is therefore retained.

Only one thread can be active within a monitor at any one time. That thread is called the monitor's *owner*. When a thread attempts to enter a monitor that is already owned, the thread is blocked, relinquishing the processor on which the thread is executing. Eventually, the owner will release the monitor, causing the least recently executed thread waiting on the monitor to be resumed in an attempt to become the owner.

A thread may wait on a condition variable. When a condition variable is created, it must be bound to some monitor. The condition variable should only be used from within that monitor. It is an error to do otherwise. A thread waiting on a condition variable releases the associated monitor as it blocks. Another thread can signal the condition variable, causing the condition variable's longest waiting thread eventually to resume. The signaller continues to own the monitor until it waits or exits, so a signalled thread, since it must reacquire the monitor, does not execute immediately upon being signalled. A signal must be regarded merely as a hint that an acceptable state had been reached at some prior point. When a waiting thread next runs again, it should check that the condition on which it waited has remained satisfied since being signalled. A thread may also broadcast on a condition variable, causing all threads waiting on that condition variable to be signalled.

The following example demonstrates monitored access to stacks. The class SynchronizedStack is defined as a sub-class of the Stack class presented earlier. Synchronized stacks have all the characteristics of their super-class, but guarantee that access to the stack is atomic by redefining pop() and push() to require the posession of an exclusive private monitor. Further, a thread trying to pop() from an empty stack blocks, and does not resume until the stack becomes non-empty.

```
        // Sub-class                    isa        Super-class
class SynchronizedStack                  :          Stack {
        //
        // Our own private data for synchronizing.
        //
        Monitor *s_monitor;                        // — for atomic access
        Condition *s_condition;                    // — reads are blocking
public:
        // Constructor to create a new stack
        Stack(int sz)
        :(sz)       // Call CONSTRUCTOR of Super-class
              {
              s_monitor = new Monitor("StackMonitor");
              s_condition = new Condition(s_monitor, "StackCondition");
              }
        void push(int newitem)
              {
              MONITOR ENTRY (s_monitor);
                           // Qualify to the super-class operation
```

```
                    Stack::push (newitem);
                    if (depth() == 1     {
                          // Signal if any could be waiting
                        s_condition->signal();
                    }
            }
        int pop()
            {
            MONITOR ENTRY(s_monitor);
            int topitem;
                while (depth() == 0)  {
                        s_condition->wait();
                        // Consider the signal only as a hint.
                        // Must check depth again.
                }
                topitem = Stack::pop();
            return topitem;
           }
    };
```

Using instances of this class, multiple threads can safely share the same stack. Furthermore, operations written to operate on the super-class Stack can just as easily operate on a SynchronizedStack.

Atomic integers

To address the common situation where one would like simply to update a counter or some other integral value within an otherwise unsynchronized region of code, PRESTO provides an atomic integer class. The class AtomicInt guarantees multiple-reader, single-writer semantics for integers by automatically enclosing their reference within a spinlock's lock() and unlock(). AtomicInt supports the full complement of integer operations (assignment, increment, decrement, etc.). An AtomicInt can be used anywhere an integer is expected:

```
    {
                AtomicInt      a;
                AtomicInt      b = 10;
                int            c = a;
                int            d;

                // w_lock -> write lock; w_unlock -> write unlock
                // r_lock -> read lock; r_unlock -> read unlock

                d = a++;       // w_lock a, a++, d = a, w_unlock a

                b += c;        // w_lock b, increment b by c, w_unlock b

                c = a + b;     // r_lock a; a' = a; r_unlock a;
                               // r_lock b; b' = b; r_unlock b;
```

```
              // c = a' + b'
a = a + b;    // r_lock a; a' = a; r_unlock a;
              // r_lock b; b' = b; r_unlock b;
              // w_lock a; a = a' + b'; w_unlock a;
```

Atomic integers are an example of how the object-oriented programming model meshes well with the needs of parallel programs. Instances of class AtomicInt are responsible for ensuring their own synchronization and providing their own access semantics in a parallel environment. Users of the class are insulated from the details of the class's implementation, and are guaranteed of its correct operation.

SYSTEM ARCHITECTURE

PRESTO exists as a run-time library on Sequent Balance and Symmetry shared-memory multiprocessors. (PRESTO will soon also be operational on the DEC SRC Firefly, an experimental prototype multiprocessor workstation).

The Sequent's operating system is Dynix, a 4.2BSD UNIX† look-alike with support for shared memory. Dynix[10] provides support for writing parallel programs, but this support is limited. The Dynix unit of execution is the UNIX process, an expensive ('heavyweight') and inflexible entity. Because the basic synchronization mechanisms are cumbersome to use, a 'parallel programming library' is provided. This library restricts the 'threadedness' of a parallel program to the number of physical processors in the system, prohibiting the design of algorithms that have hundreds (or even thousands) of independent threads of execution. Even if the parallel programming library were redesigned to remove this restriction, the performance of the basic system primitives would seriously limit the ways in which parallelism could be used. (We must emphasize that these limitations are *not* unique to Dynix and the Sequent, and that on balance we are delighted with the Sequent system.)

The basic role of the PRESTO run-time system is to map user's threads onto physical processors and to provide access to a global shares memory in which all objects reside. In the case of a system like Dynix, PRESTO maps threads onto Dynix processes, relying on the Dynix kernel to complete the mapping onto a physical processor. Although there are two levels of indirection required, a Dynix process can be permanently bound to a physical processor, so the second level of indirection is done only once. All details of the mappings are invisible to the PRESTO programmer.

The architecture of PRESTO adheres to the threaded object model described earlier. The system maintains a single scheduler object. The scheduler object keeps track of all threads that are runnable but not yet running. A thread becomes runnable when first started within an object, or when resumed by a synchronization object after blocking. Each processor in the system is represented by a processor object. There may be more processor objects than processors, but this is not the intention. One scheduler thread runs within each processor object, and that thread's only activity is to request runnable threads from the scheduler object. When a scheduler thread obtains a runnable thread from the scheduler object, the scheduler thread stops, and the processor on which the scheduler thread was running begins running the now-runnable thread. When the newly-running thread blocks or terminates, the scheduler thread is resumed and continues to check for more runnable threads.

---

† UNIX is a trademark of AT&T, Bell Laboratories.

Simultaneous requests to the scheduler object from multiple scheduler threads are synchronized so that no thread can be scheduled on more than one processor at any instant. However, a thread may execute on different processors at different times. Migration occurs only if a thread is blocked and then resumed at some later time when some other processor is idle. Scheduler threads are an exception to this — they *never* migrate. A scheduler thread runs only on the processor for which it is scheduling. Figure 1 illustrates how the scheduler object, procesor objects and physical CPUs are related. Because these objects interact with one another only through their operations, each can be easily replaced or modified without affecting the others. For example, the scheduler object could be changed to maintain multiple priority queues for threads rather than a single runnable queue. Since scheduler threads interact with the scheduler only through a GetAThread() operation, they would remain unaffected by the change.

The PRESTO scheduler eventually halts when there are no longer any runnable or running threads. At this point, all existing synchronization objects are destroyed. If any one of them indicates a waiting thread, the system declares deadlock and displays the state of all interminably blocked threads. Because a thread waiting on a spinlock is still technically executing, this very simple criterion for detecting deadlock fails if one or more threads are waiting for a spinlock to become free. More sophisticated halting semantics have been implemented through a redefinition of the scheduler. For example, a message-based discrete event simulation scheduler has been built that resolves deadlock arising from circular message dependencies when no threads are runnable.

PROGRAM STRUCTURE

In PRESTO, a user's parallel program consists of a set of class definitions for objects used in the program, and a set of implementation routines that define the operations for each class. In addition, the programmer must provide one operation for a system defined class called Main:

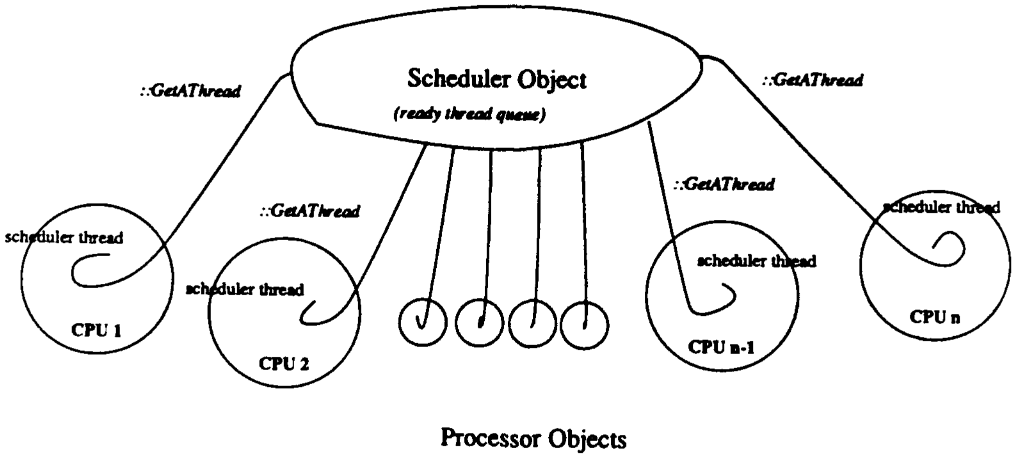


Figure 1. PRESTO components

```
//
//Programmer supplied
//
int
Main::main()
{
	//
	// Called once the system has started. There
	// will be at least one thread started in this
	// routine.
	//
}
```

The programmer links his code with the PRESTO library and obtains an executable program. The function main() required by the UNIX loader is already provided by the library. This routine creates an object of class Main and starts at least one thread in the operation Main::main() for that object.

The programmer may also provide two other operations, Main::init() and Main::done(). Main::init(), if provided, is called before the system begins executing; it can be used to override certain system default parameters such as the number of processors to use. Main::done(), if provided, is called when there are no more runnable threads, allowing PRESTO programs to clean up after themselves. Once running in Main::main(), the system is under the control of the programmer.

SOME EARLY PERFORMANCE FIGURES

This section presents early performance measurements for PRESTO. All figures represent measurements taken from a Sequent Balance 21000 with ten 32032 processors. As a baseline, a processor can do a null procedure call and return in approximately 15 μs, and can execute a single iteration of a for-loop in 7 μs.

Program performance

Figure 2 illustrates the performance of PRESTO when running a matrix multiplication algorithm over varying numbers of processors. The problem was decomposed equally among as many threads as there were processors, and each thread ran independently of the others. The two optimal curves are based on the performance of the algorithm when run with *n* threads on *n* processors. These are clearly best-case examples, since the scheduling and synchronization costs imposed by the algorithm are essential zero. Nevertheless, it shows that an optimal breakdown of an optimal problem can yield nearly optimal results under PRESTO. An implementation of the same algorithm directly on top of Dynix performs identically. This is not surprising since the processor speed is the same, and neither PRESTO nor Dynix are doing anything to assist (or hinder) the computation.

A difference does exist between the optimal and measured curves, and this same difference exists in the Dynix implementation of the same algorithm. It is primarily attributable to the start-up costs of the program and of initializing the processors. The matrices to be multiplied must first be initialized. Although data initialization could

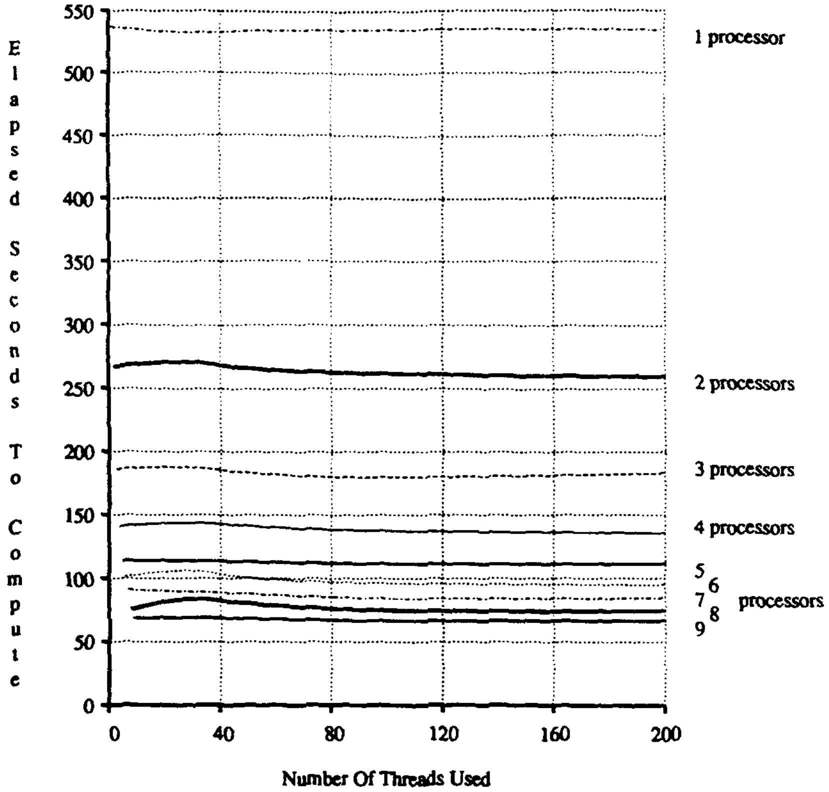


Figure 3. Matrix multiplication thread decomposition (200×200)

significant hardship. As noted, a key goal of PRESTO is to decrease the cost of parallelism so that the problem structure, rather than the underlying system, can be allowed to determine the way in which parallelism is used.

To demonstrate that this key goal was achieved, Figure 3 illustrates the performance of the PRESTO matrix multiplication algorithm as the number of threads is allowed to become very large (as many as 200 threads working on a 200×200 matrix). The Figure demonstrates that fine granularity using many threads can be inexpensive with PRESTO — an important advantage over many existing parallel programming systems. In PRESTO, the cost of several hundred threads is not much more than the cost of a few threads (except for the small cost of first scheduling each thread). Thus, in PRESTO, threads can be used as a 'program structuring' tool.

An interesting characteristic of Figure 3 is the small hump at around 25 threads for several of the curves. The hump is due to a decomposition of the problem inappropriate to the number of processors used. Let E be the execution time of one thread on one processor, n be the number of processors available and T be the number of threads over which a computation is divided. Since E is the same for all threads, execution essentially proceeds in lock-step. At each step, T/n threads complete, except that when n is not a factor of T, it is not possible to evenly distribute the final $T \bmod n$ threads over the n available processors. Consequently, the total computation time, t_c, is

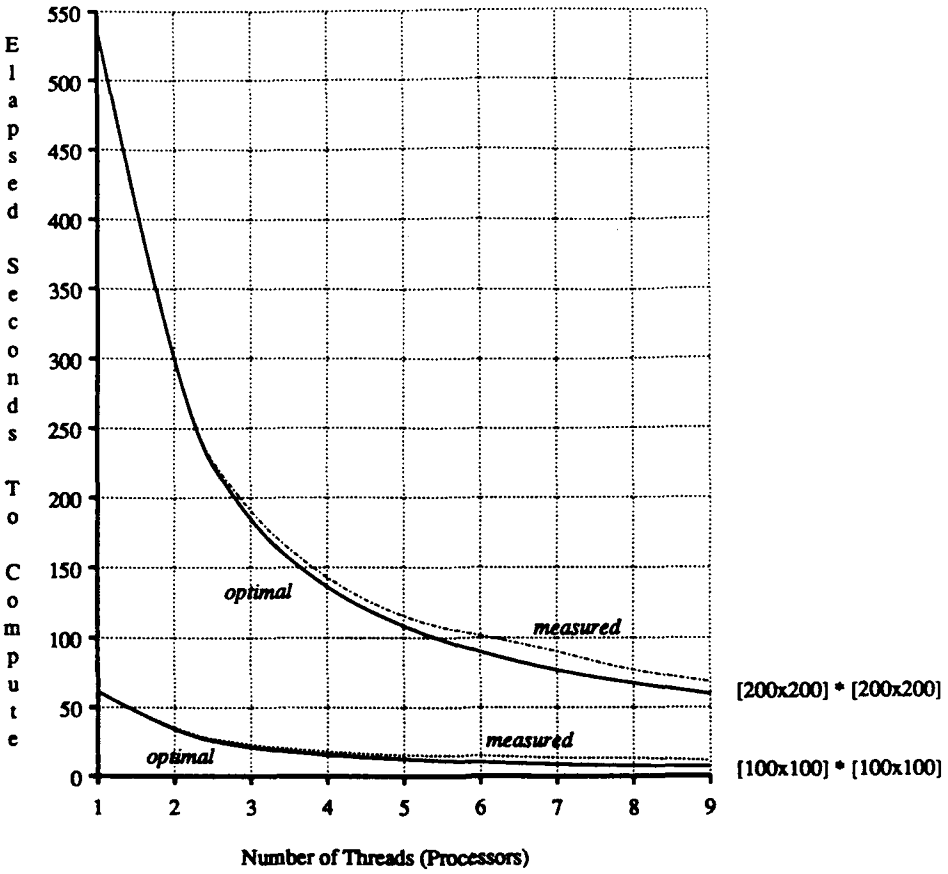


Figure 2. Matrix multiplication speed-up

be done in parallel, the program represented in Figure 2 does not do so. The data initialization costs are reflected only in the measured curves, not in the optimal ones. In addition to program initialization, PRESTO itself must be initialized. The time to initialize and begin executing on nine processors is much greater than for a single processor (the cost is roughly 55 ms per processor), but the total lifetime of the computation is much shorter. Consequently, the percentage of time doing work unrelated to the multiplication algorithm increases with the number of processors, and this appears as a break from the optimal curve when the total computation time gets very small. For longer-running computations, the initialization effects disappear.

The cost of threads

In Figure 2, the matrix multiplication algorithm was designed so that the number of threads was equal to the number of available processors. This is in some sense 'optimal' from a performance point of view.

In the case of matrix multiplication, designing an algorithm that is 'parametrized' by the number of processors is straightforward. In other problem settings, though, there may be a 'natural' decomposition of the problem into threads of control, and 'warping' this decomposition onto a specific number of processors may impose a

$$t_c = \left\lceil \frac{T}{n} \right\rceil \times E$$

When T is small and E is large (as it is with only 25 threads working on a 200×200 matrix), this 'tail effect' can be quite pronounced and manifests itself as the hump seen in Figure 3. The hump is largest when $T \bmod n$ is small, but non-zero. For example, when n is eight, seven processors are wasted during the final phase of the computation as only one thread remains to execute. With nine processors though, seven threads execute during the final phase, leaving only two processors idle. As would be expected, 25 threads on five processors produces no hump.

This phenomenon of some processors idling while others work is called *starvation*. When processors execute in lock-step, the amount of wasted processing time due to starvation effects is $E \times n_s$ where n_s is the number of processors suffering from starvation. Even when processors execute asynchronously, unless all threads terminate concurrently, there must come a time near the end of a computation when there are fewer runnable threads than processors. Starvation can even become a factor when the 'optimal' number of threads is used but some delay exists between the starting time of the first thread and that of the last. Clearly, the negative impact of starvation diminishes with decreasing E. For fixed size problems, E decreases with increasing T. If the overhead due to a large T can be offset by preventing the degrading effects of starvation, appreciable performance benefits can be realized through very fine grained decomposition. PRESTO makes this possible.

Towards cheaper threads

The construction of a thread's call-stack is a significant contributor to thread creation cost. To mitigate this in PRESTO, threads are reclaimed upon termination for possible reuse. When the programmer requests a new Thread, the system checks if any reclaimed threads are available. If so, a new thread is not created; the reclaimed one is reinitialized and returned to the programmer. If not, a thread template is created and marked as incomplete. The thread template can be manipulated in the same ways as a complete thread. Eventually, the thread template will attempt to execute for the first time. When this happens, the reclaim pool is checked again. Only if it is empty the second time is an entirely new thread created and initialized with the values stored in the template.

This aggressive design, which is totally transparent to the programmer, significantly

Table II. Thread creation and destruction costs

Processors	Threads created and destroyed	Elapsed time (s)
1	100,000	44·0
2	100,000	28·2
3	100,000	21·6
4	100,000	18·1
5	100,000	15·8
6	100,000	14·0
7	100,000	12·8
8	100,000	11·8

reduces the cost of threads in situations where a number of threads are started simultaneously and run to termination without blocking: only as many call-stacks will be allocated as there are processors. A peculiar side-effect is that it is difficult to talk in a meaningful way about the 'cost of thread creation' in PRESTO, since this cost depends upon the style of use.

Table II shows the time to create and destroy PRESTO threads when every thread can be reclaimed. The table demonstrates two points. First, thread creation is relatively inexpensive. For the single processor case, the average time to create and destroy a thread is about 440 μs. Secondly, the rate at which threads can be created, while not linear with the number of processors, is much better than constant. The non-linearity arises because all threads originate from and return to a common pool, and access to this pool can be a bottleneck as locking must occur. Practically though, bursty periods of thread creation are usually the result of a single thread co-ordinating the activities of the new threads, so high contention is unlikely.

The cost of synchronization

There are two types of synchronization costs in a parallel program: non-competitive and competitive. A non-competitive cost is incurred whenever a thread accesses a synchronization object and is able immediately to become its owner. The programmer pays the competitive cost whenever a thread must wait for some other thread to relinquish ownership, or when the thread blocks on a condition variable. The competitive cost always includes, and is therefore greater than, the non-competitive cost. Table III shows these costs for four different synchronization operations. Only the overhead involved in actually blocking and unblocking a thread is reflected in these figures.

The top half of the table represents non-competitive synchronization overhead, and the bottom, competitive. Lock_test, monitor_test, spin_test and atom_test show the times required for threads to acquire and release a simple relinquishing lock, monitor, spinlock and atomic integer, respectively. The 21 μs required to use a spinlock is due to the slowness of the hardware atomic locks available on the Sequent. Since spinlocks serve as the basis for all other synchronization primitives, their lack-lustre performance negatively influences the other timings. When atom_test is run with two processors,

Table III. Synchronization costs

	Benchmark	Processors	Threads	Iterations	Elapsed time (s)	Average time (μs)
Non-competitive	lock_test	1	1	1,000,000	94·9	94·9
	monitor_test	1	1	1,000,000	153	153
	spin_test	1	1	1,000,000	21·2	21·2
	atom_test	1	1	1,000,000	25·7	25·7
Competitive	lock_test	2	2	1,000,000	158·1	158·1
	monitor_test	2	2	1,000,000	238·3	238·3
	switch_test	1	2	100,000	123·9	1239
	switch_test	2	2	100,000	73·1	731
	atom_test	2	2	1,000,000	27·7	27·7

the elapsed time increases slightly over the single-processor case. This is due to an optimization for spinlocks (on which atomic integers are based) biased towards non-competitive acquisition. When the optimization is removed, one processor performs no better than two.

The time required for two threads to switch back and forth on a condition variable is shown for both one and two processors in switch_test. Each thread enters the monitor, signals a condition variable, and then waits on that condition variable, relinquishing the monitor (and the processor on which it is running). Although only one thread can be active in the monitor at any instant, two processors perform substantially better than one. The reason is that the context switch times for the waiting and signalled threads can be overlapped so that one thread can be switched in while the other is being switched out. With a single processor, this is not possible.

CONCLUSIONS

PRESTO is both a production system for use in writing everyday parallel programs and a flexible research tool with which various scheduling, synchronization and granularity issues can be explored.

The former goal is met by joining the classical notions of concurrent programming with the powerful concepts of object-oriented design. Objects can be made completely responsible for their own execution, as well as modification and presentation. This relieves the user of an object from concern about potential misuse in a parallel environment. By exploring the class mechanism of the language, programmers can derive parallel, inherently safe objects (such as a synchronized stack) from simpler, well-understood, sequential versions. A key point, emphasized in this paper, is that the performance of PRESTO's primitives is sufficiently good that the 'natural' decompositions of problems, rather than artificial constraints imposed by the system, can be the determining factor in the structure of parallel algorithms.

The utility of PRESTO as a research vehicle arises from its underlying structure. A system component (the scheduler, a processor, even a thread) can be redefined through inheritance without affecting the other components.

PRESTO is not a toy. It is the current system of choice for parallel programming at the University of Washington. Certain applications have been built on top of the 'default' Mesa-like environment; for example, a parallel solution package for queueing network performance models. Other applications have customized certain aspects of PRESTO, taking advantage of its 'open' design; a parallel Othello program involves a new PRESTO scheduler; an instrumentation package for parallel programs involves an extension of threads to include monitoring capabilities.

ACKNOWLEDGEMENTS

We would like to thank Kenneth Almquist, Tom Anderson, Jeff Chase, and David Wagner for their user-view feed-back on the design and implementation of PRESTO. They, along with Ellen Ratajak, Doug Comer, and the referees, provided many helpful comments concerning this paper. Our work is supported by the U.S. National Science Foundation (Grants No. CCR-8619663, CCR-8700106, and CCR-8703049), the Naval Ocean Systems Center, US WEST Advanced Technologies, the Washington Technology Center, the USENIX Association, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

REFERENCES

1. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, March 1986.
2. G. T. Almes, A. P. Black and E. D. Lazowska, 'The Eden system: a technical review', *IEEE Trans. Software Engineering*, **SE–11**, 43–58 (1985).
3. E. Jul, H. Levy, N. Hutchinson and A. Black, 'Fine-grained mobility in the Emerald system', *ACM TOCS*, **6**,(1) 109–133 (1988).
4. B. N. Bershad, E. D. Lazowska, H. M. Levy and D. Wagner, 'An open environment for building parallel programming systems', *Proc. ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, July 1988.
5. *Modula2+ Reference Manual*, Digital Equipment Corporation, April 1986.
6. T. W. Doeppner Jr. and Alan J. Gebele, 'C++ on a parallel machine', *Report CS–87–26*, Department of Computer Science, Brown University, November 1987.
7. C. A. R. Hoare, 'Monitors: an operating system structuring concept', *Communications of the ACM*, **17**, (10), 549–557 (1974).
8. B. W. Lampson and D. D. Redell, 'Experiences with processes and monitors in Mesa', *Communications of the ACM*, **23**, (2), 104–117 (1980).
9. J. G. Mitchell, W. Maybury and R. Sweet, 'MESA language manual', *Technical Report CSL-79-3*, Xerox Palo Alto Research Center, April 1979.
10. S. S. Thakkar, P. Gifford and G. Fielland, 'Balance: a shared memory multiprocessor', *Proceedings, 2nd International Conference on Supercomputing*, Santa Clara, May 1987.

Final Scientific and Technical Report

The Performance of Parallel Computer Systems

Delivery Order No. 0009, Contract No. N66001-87-D-0136

Edward D. Lazowska and John Zahorjan
Department of Computer Science and Engineering
University of Washington

November 1989

This report is a summary of the research conducted under this project, which began on May 1 1988 and concluded on September 30 1989.

Roughly 30 technical reports have been submitted to NOSC covering various specific developments. In this report, we will summarize each area of investigation, and we will include abstracts of the relevant publications. Thus, the main purpose of this report is to provide perspective – to make it clear that the large number of technical reports that we have previously submitted are in fact related to the pursuit of a small number of major themes.

We note that in addition to NOSC, our research is sponsored by the National Science Foundation, Digital Equipment Corporation, U S WEST Advanced Technologies, and the Washington Technology Center.

Motivation

The objective of our work is to develop tools and techniques that facilitate the production of high-performance parallel programs for medium-scale shared-memory multiprocessors.

To motivate this work, we shall spend just a moment reviewing four central issues: the reasons for studying parallel computing in general and medium-scale multiprocessing in particular, the potential cost/performance benefits of medium-scale multiprocessing, the track record to date in achieving this potential, and the complexities that make system design difficult in this environment.

High-performance computing is of critical importance. Parallel systems hold great promise for cost-effectively providing this high performance. Medium-scale multiprocessing is a natural area for study, because these systems are available today, and raise a host of difficult problems, most of which are relevant to more advanced architectures as well.

The potential cost/performance benefits of medium-scale multiprocessing are indicated by the fact that the DEC Systems Research Center, which designed and implemented the Firefly experimental prototype multiprocessor workstations that we use in our research, estimates that the production cost of a five processor workstation is only 15% greater than the production cost of a uniprocessor workstation that uses the same technology. Thus, the Firefly, which is by no means an aggressive design, offers a *potential* cost/performance advantage of greater than 4:1.

The challenge, of course, lies in achieving this potential. The "bread and butter" of Sequent Computer Systems, manufacturer of the Symmetry and Balance multiprocessor systems (we use a 20-processor Symmetry in our research), is customers who desire inexpensive sequential UNIX cycles, which the Sequent systems deliver with great success. The small proportion of Sequent customers whose objective is to run production parallel applications face much greater challenges. Sequent tells the story of Teradyne, a company that sells circuit simulation software. Teradyne set out to parallelize its existing software, with the objective of selling a turnkey "circuit simulation engine" based on an 8-processor Sequent. After months of effort, Teradyne's engineers hit the "go" button only to discover that their parallel simulator took twice as long to run on an 8-processor system as their sequential simulator had taken on a uniprocessor. (The good news is that nearly linear speedup was achieved through extensive "performance debugging"; the bad news is that this is by no means an isolated incident.)

As just one example of a complexity that makes the design of parallel systems difficult, consider the choice between spinning (busy waiting) and blocking (relinquishing the processor) to wait for an event. The appropriate choice between spinning and blocking depends on the relationship of the expected spin time to the context switch time. This choice is not always clear, and a mistake can have major performance implications. For example, one field test release of the Sequent's DYNIX operating system included the substitution of blocking for spinning in a single routine as a "performance enhancement". Under high loads, this change in fact caused a severe performance degradation, something that was first noticed by a Sequent competitor and used as the basis of an advertising campaign.

This brief motivation hopefully suffices to indicate the breadth, depth, and importance of the problem area that we are tackling. Looking in a bit more detail, our technical objectives can be divided into six categories, discussed in the sections that follow.

Operating system support for parallel computing

The goal of this research is to provide efficient primitives to support parallel computing, so that the programmer can use parallelism in a manner dictated by the nature of the application rather than by the cost of the primitives.

Zahorjan, Lazowska, and Eager have analytically examined lock acquisition strategies – in particular, the tradeoffs between spin-waiting and blocking [Zahorjan, Lazowska & Eager 1988]. They have studied the degradation of spin-waiting as the variability in lock holding times increases (this variability might be due to multiprogramming or to data dependencies), and they have found relatively simple scheduling rules that can avoid this degradation. They have also explored in more detail the interaction between scheduling discipline and spin overhead [Zahorjan, Lazowska & Eager 1989].

Anderson, Lazowska, and Levy have experimentally examined data structure alternatives for thread management [Anderson, Lazowska & Levy 1989] and algorithm alternatives for spin-waiting [Anderson 1989]. With Bershad, they have provided a comprehensive overview of thread management [Anderson et al. 1989].

Anderson, Bershad, Lazowska, and Levy have developed a highly-efficient protected procedure call mechanism [Bershad et al. 1989]. The assertion is that using the network IPC or RPC mechanism for cross-address-space calls on a single machine (as is done, e.g., in Mach, Topaz, and V) fails to "optimize the common case", with significant performance and structure repercussions. Major improvements have been attained.

McCann and Zahorjan are investigating scheduling strategies for multiprogrammed shared memory parallel systems [Zahorjan & McCann 1989]. They have proposed a number of policies wherein the system scheduler allocates processors to jobs and the application itself schedules threads on those processors. An initial simple analysis of the policies has been performed and work is progressing on more detailed simulations and eventual implementation of promising strategies on our Sequent multiprocessor.

Programming support for parallel and parallel/distributed computing

In this research we explore the software support that sits between the operating system and the programmer, and that facilitates the development of high-performance parallel applications.

Bershad, Lazowska, and Levy developed Presto, an object-based "toolkit" for parallel programming which is based on C++ [Bershad 1988; Bershad, Lazowska & Levy 1988; Bershad et al. 1988]. Presto attacks two key problems in parallel programming: the high cost of the primitives supporting parallelism (discussed in the previous section), and the fact that most parallel programming systems present the programmer with a rigid "model" of parallelism that may not be appropriate to the specific application domain at hand. In Presto, the programmer first extends the base system to provide direct support for the application domain at hand, and then programs the specific application.

Chase, Amador, Lazowska, Levy, and Littlefield are developing Amber, which can be thought of as "distributed Presto" [Chase et al. 1989]. The assertion is that networks of medium-scale multiprocessors will soon be commonplace, and that it will be desirable to program them as a single system, using hardware coherence within each multiprocessor and software coherence (based on the Amber object model) between them. Amber is being prototyped on our DEC SRC Firefly multiprocessor workstations.

Parallel discrete-event simulation

Parallel simulation is a natural application area for three reasons: simulation is widely used in other aspects of our research, parallel simulation provides a realistic test of our system and programming support for parallel computing, and parallel simulation is of intrinsic interest.

Wagner and Lazowska have designed and built Synapse, a system based on Presto that aggressively exploits shared memory to speed up "conservative" parallel simulation [Wagner, Lazowska & Bershad 1989]. They have also examined the particular application domain of queueing network simulations, and have discovered a number of successful optimizations [Wagner & Lazowska 1989]. Further details on this work are contained in Wagner's Ph.D. dissertation [Wagner 1989].

While Wagner and Lazowska's exploration of parallel simulation is largely experimental, Lin, Baer, and Lazowska have taken a more analytic approach to studying both conservative and optimistic parallel simulation. In the conservative realm, they began by examining the specific application domain of multiprocessor cache coherence algorithm simulation. They derived a general technique and showed how performance can be improved by tailoring this technique to specific cache coherence protocols [Lin, Baer & Lazowska 1989; Lin, Lazowska & Baer 1989; Eggers, Lazowska & Lin 1989]. They have examined the efficient parallel simulation of systems with no lookahead [Lin, Lazowska & Baer 1990], discovered novel ways of exploiting lookahead when it does exist [Lin & Lazowska 1989c], and developed techniques for deriving improved lookahead estimates in certain simulations [Lin, Lazowska & Baer 1989b]. In the optimistic realm, Lin and Lazowska have studied situations under which this approach yields the optimal simulation [Lin & Lazowska 1990], have determined the optimal checkpoint interval for optimistic simulation [Lin & Lazowska 1989a], and have developed improvements to the Time Warp rollback mechanism [Lin & Lazowska 1989b].

Parallel performance analysis and performance tools

Of course, performance lies at the heart of everything that has been discussed thus far. However, we are conducting both general and specific studies in the specific area of parallel system performance analysis.

Eager, Zahorjan, and Lazowska have attempted to determine appropriate abstractions for viewing parallel software running on parallel hardware from the performance point of view [Eager, Zahorjan & Lazowska 1989]. The objective of this work is to develop a framework for building measurement tools and modelling tools for parallel systems.

Several simple measurement tools have been constructed – by Anderson on the Sequent and by Bershad on the Firefly. More work in this area remains to be done. The Quartz tool, recently developed, represents a significantly novel approach to the instrumentation of parallel application programs for the purpose of performance tuning [Anderson & Lazowska 1989].

The development of useful performance tools is driven by applications. Our work in parallel discrete-event simulation is one such application. Another is an experimental study of parallel dynamic programming [Almquist, Anderson & Lazowska 1989].

Design of parallel computer memory interconnects

Mizrahi, Baer, Lazowska, and Zahorjan have designed and analyzed an architecture called the Memory Hierarchy Network, an interconnection network with memory at the nodes and data migration in response to reference patterns [Mizrahi et al. 1989a, 1989b].

Vernon, Lazowska, and Zahorjan have developed highly efficient analytic tools for studying cache design questions [Vernon, Lazowska & Zahorjan 1988].

Distributed and heterogeneous computer systems

We maintain a strong interest in various issues related to distributed and heterogeneous computer systems.

In the heterogeneous system domain, our major recent focus has been on file systems and remote procedure call mechanisms. Pinkerton, Lazowska, Notkin, and Zahorjan have developed a design for a heterogeneous remote file system that offers a number of significant advantages over alternative designs, in particular FTAM [Pinkerton et al. 1989]. Chung, Gosney, Lazowska, and Notkin have developed an extension to our previously-developed heterogeneous remote procedure call facility that provides a multi-language capability [Chung et al. 1989a].

In the distributed system domain, Chung, Lazowska, Notkin, and Zahorjan have studied the performance implications of various design alternatives for remote procedure call stubs – for example, the choice between compiled and interpreted stubs [Chung et al. 1989b].

Neuman and Lazowska are exploring mechanisms to provide "customized views" of very large distributed systems [Neuman 1989]. This work begins from the observation that while each user needs transparent access to the entirety of such a system, each user also needs to be able to easily construct a view of the system that highlights those things that are relevant and suppresses those things that are not. The approach being taken is called the "virtual system" approach; a file system prototype built using this approach is currently under construction.

References

[Almquist, Anderson & Lazowska 1989]
Kenneth Almquist, Richard J. Anderson, and Edward D. Lazowska. The Measured Performance of Parallel Dynamic Programming Implementations. *Proc. 1989 International Conference on Parallel Processing*, August 1989.

[Anderson 1989]
Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *Proc. 1989 International Conference on Parallel Processing*, August 1989. To appear, *IEEE Transactions on Distributed and Parallel Systems*.

[Anderson et al. 1989]
Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Thread Management for Shared-Memory Multiprocessors. Technical Report 89-10-02, Department of Computer Science and Engineering, University of Washington, October 1989. Submitted for publication.

[Anderson & Lazowska 1989]
Thomas E. Anderson and Edward D. Lazowska. Quartz: A Tool for Tuning Parallel Program Performance. Technical Report 89-09-05, Department of Computer Science and Engineering, University of Washington, September 1989. Submitted for publication.

[Anderson, Lazowska & Levy 1989]
Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1989; forwarded by the program committee as an award paper to to *IEEE Transactions on Computers*, to appear.

[Bershad 1988]
Brian Bershad. Presto User's Guide. Technical Report 88-01-04, Department of Computer Science, University of Washington, January 1988.

[Bershad et al. 1989]
Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. To appear, *ACM Transactions on Computer Systems*, February 1990; forwarded by the program committee as an award paper to *ACM Transactions on Computer Systems, to appear*.

[Bershad, Lazowska & Levy 1988]
Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice & Experience 18*,8, August 1988, pp. 713-732.

[Bershad et al. 1988]
Brian N. Bershad, Edward D. Lazowska, Henry M. Levy, and David B. Wagner. An Open Environment for Building Parallel Programming Systems. *Proc. ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, July 1988.

[Chase et al. 1989]
Jeffrey S. Chase, Franz Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. To appear, *Proc. 12th ACM Symposium on Operating Systems Principles*, December 1989.

[Chung et al. 1989a]
Sung K. Chung, Kimiko Gosney, Edward D. Lazowska, and David Notkin. Multi-Language Support for Heterogeneous Remote Procedure Call. Technical Report 89-10-09, Department of Computer Science and Engineering, University of Washington, October 1989. Submitted for publication.

[Chung et al. 1989b]
Sung K. Chung, Edward D. Lazowska, David Notkin, and John Zahorjan. Performance Implications of Design Alternatives for Remote Procedure Call Stubs. *Proc. 9th International Conference on Distributed Computing Systems*, June 1989.

[Eager, Zahorjan & Lazowska 1989]
Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, March 1989.

[Eggers, Lazowska & Lin 1989]
Susan J. Eggers, Edward D. Lazowska, and Yi-Bing Lin. Techniques for the Trace-Driven Simulation of Cache Performance. Invited for *1989 Winter Simulation Conference*, December 1989.

[Lin, Lazowska & Baer 1989]
Yi-Bing Lin, Edward D. Lazowska, and Jean-Loup Baer. Parallel Trace-Driven Simulation of Multiprocessor Cache Performance: Algorithms and Analysis. Invited chapter in *Progress in Simulation*, Ablex Publishing, 1989.

[Lin, Baer & Lazowska 1989]
Yi-Bing Lin, Jean-Loup Baer, and Edward D. Lazowska. Tailoring a Parallel Trace-Driven Simulation Technique to Specific Multiprocessor Cache Coherence Protocols. *Proc. 1989 Distributed Simulation Conference*, March 1989.

[Lin & Lazowska 1990]
Yi-Bing Lin and Edward D. Lazowska. Optimality Considerations for Time Warp Parallel Simulation. To appear, *Proc. SCS Multiconference on Distributed Simulation*, January 1990.

[Lin, Lazowska & Baer 1990]
Yi-Bing Lin, Edward D. Lazowska, and Jean-Loup Baer. Conservative Parallel Simulation for Systems with No Lookahead Prediction. To appear, *Proc. SCS Multiconference on Distributed Simulation*, January 1990.

[Lin & Lazowska 1989a]
Yi-Bing Lin and Edward D. Lazowska. The Optimal Checkpoint Interval in Time Warp Parallel Simulation. Technical Report 89-09-04, Department of Computer Science and Engineering, University of Washington, September 1989. Submitted for publication.

[Lin & Lazowska 1989b]
Yi-Bing Lin and Edward D. Lazowska. A Study of Time-Warp Rollback Mechanisms. Technical Report 89-09-07, Department of Computer Science and Engineering, University of Washington, September 1989. Submitted for publication.

[Lin & Lazowska 1989c]
Yi-Bing Lin and Edward D. Lazowska. Exploiting Lookahead in Parallel Simulation. Technical Report 89-10-06, Department of Computer Science and Engineering, University of Washington, October 1989. Submitted for publication.

[Mizrahi et al. 1989a]
Haim E. Mizrahi, Jean-Loup Baer, Edward D. Lazowska, and John Zahorjan. Introducing Memory into the Switch Elements of Multiprocessor Interconnection Networks. *Proc. 16th International Symposium on Computer Architecture*, May 1989.

[Mizrahi et al. 1989b]
Haim E. Mizrahi, Jean-Loup Baer, Edward D. Lazowska, and John Zahorjan. Extending the Memory Hierarchy into Multiprocessor Interconnection Networks: A Performance Analysis. *Proc. 1989 International Conference on Parallel Processing*, August 1989.

[Neuman 1989]
B. Clifford Neuman. The Virtual System Model for Large Distributed Operating Systems. Technical Report 89-01-07, Department of Computer Science University of Washington, April 1989.

[Pinkerton et al. 1989]
C. Brian Pinkerton, Edward D. Lazowska, David Notkin, and John Zahorjan. A Heterogeneous Distributed File System. Technical Report 88-08-08, Department of Computer Science, University of Washington, August 1988, revised October 1989. Submitted for publication.

[Vernon, Lazowska & Zahorjan 1988]
Mary K. Vernon, Edward D. Lazowska and John Zahorjan. An Efficient and Accurate Performance Analysis Technique for Multiprocessor Snooping Cache Consistency Protocols. *Proc. 15th International Symposium on Computer Architecture*, May 1988.

[Wagner 1989]
Conservative Parallel Discrete Event Simulation: Principles and Practice. Technical Report 89-09-03, Department of Computer Science and Engineering, University of Washington, September 1989 (Ph.D. thesis).

[Wagner & Lazowska 1989]
David B. Wagner and Edward D. Lazowska. Parallel Simulation of Queueing Networks: Limitations and Potentials. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1989.

[Wagner, Lazowska & Bershad 1989]
David B. Wagner, Edward D. Lazowska, and Brian N. Bershad. Techniques for Efficient Shared-Memory Parallel Simulation. *Proc. 1989 Distributed Simulation Conference*, March 1989.

[Zahorjan, Lazowska & Eager 1988]
John Zahorjan, Edward D. Lazowska, and Derek L. Eager. Spinning Versus Blocking in Parallel Systems with Uncertainty. *Proc. International Symposium on Performance of Distributed and Parallel Systems*, December 1988.

[Zahorjan, Lazowska & Eager 1989]
John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. Technical Report 89-07-03, Department of Computer Science and Engineering, University of Washington, July 1989. Submitted for publication.

[Zahorjan & McCann 1989]
John Zahorjan and Cathy McCann. Processor Scheduling in Shared Memory Multiprocessors. Technical Report 89-09-17, Department of Computer Science and Engineering, University of Washington, September 1989. Submitted for publication.

Programming Support for Parallel and Parallel/Distributed Computing

The PRESTO User's Manual

Brian N. Bershad

University of Washington
Department of Computer Science
Seattle, Washington 98195
(206) 545-2675

January 1988

ABSTRACT

PRESTO is an environment for writing object-oriented parallel programs in the C++ programming language. It makes few assumptions about the behavior of these programs and the models of paralellism to which they adhere. This paper describes the basic PRESTO primitives and provides examples of their use. A simple programming environment based on Mesa monitors and threads is provided as part of the standard PRESTO distribution, but this may easily (and efficiently) be abandoned in preference to environments based on other models.

1. Introduction

PRESTO is an object-oriented parallel programming environment for shared-memory multiprocessors. The system provides the programmer with a set of basic classes useful for writing parallel programs in the C++ programming language. These include threads for concurrency, and locking mechanisms for synchronization. The structure of these more primitive classes and the basic PRESTO run-time environment are sufficient for writing many types of parallel programs. Nevertheless, the system is structured so that the programmer may extend, modify, or completely replace arbitrary pieces of the environment.

This manual is divided into two parts. The first addresses the mechanics of writing PRESTO programs using the structures provided by the system. Programming examples and short explanations about how the system works are also given. The second part discusses in more depth the internals of PRESTO and how to modify the system to meet the needs of specific applications. The reader is expected to have a good understanding of object-oriented programming concepts in general, and the C++ programming language in particular. To help clarify explanations, class definitions and code fragments are presented throughout this document. A more thorough motivation for the system, its structures, and its relationship to parallel programming in general can be found in [1] and [2].

2. Information For PRESTO Programmers

2.1. Parallel Programming On Top Of UNIX

A UNIX process provides a single thread of control within an address space. To the UNIX kernel, that process is the smallest schedulable entity. Unfortunately, relying directly on heavyweight UNIX processes for building parallel applications can be severely restrictive. Problems include the high cost of context switching, the limited number of simultaneous threads possible, and difficulty in sharing many types of resources between UNIX processes.

This work is supported by the National Science Foundation under Grants No. DCR-8420945, CCR-8619663, and CCR-8703049, and by grants from the Naval Ocean Systems Center, US WEST Advanced Technologies, the Washington Technology Center, the USENIX Association, and Digital Equipment Corporation's Systems Research Center and External Research Program.

PRESTO works by considering UNIX processes as physical processors. When a PRESTO program begins, some number of UNIX processes are created. A PRESTO thread can be scheduled to execute within any one of these processes. Since all PRESTO objects live in a single address space, that address space can be shared amongst all the UNIX processes, allowing a thread to move between UNIX processes. Although this structure is hidden from the PRESTO programmer (who sees only threads and a single address space), understanding how the system works can be useful during debugging. Further, understanding will make clearer the reasons for some of the system's current limitations, which .will be discussed in this document as they arise in context.

On true multiprocessors, PRESTO creates multiple UNIX processes in which to run threads. On uniprocessors, all threads are scheduled within the context of a single UNIX process. Again though, this distinction is generally transparent to the programmer who may construct applications without concern for the underlying processor structure.

2.2. C++ On A Sequent

C++ on the Sequent multiprocessor is almost identical to generic C++. The Sequent C compiler understands the two storage class identifiers `shared` and `private`. Since the latter conflicts with a C++ keyword, they have been changed for the C++ compiler to be `shared_t` and `private_t`[1]. For example,

```
static shared_t int x = 100;            // declare x to be static shared
extern private_t Thread* thisthread;    // thisthreads is private to each processor
```

By default, the loader assumes that declarations not specifying the storage class are to be private and not shared across processors. Since all PRESTO objects live in a single, shared address space, there should be no connection between a physical processor and an object. The programmer must either ensure that the `-Y` flag is given to the loader (to change the default behavior), or explicitly declare all static objects as shared (`shared_t`). Both methods are recommended as they ensure that the right thing always happens (the loader will complain if the storage class for an object is inconsistently declared). Failure to properly declare the storage class will cause strange things to happen when a static object is referenced from more than one processor. The most common types of PRESTO programming errors result from misusing the global shared memory. As a rule, PRESTO programs should almost never use private objects since threads can migrate between processors. PRESTO programs written on the Sequent can be recompiled directly on other machines, where the storage class specifiers are pre-processed away.

2.3. PRESTO Components

PRESTO consists of a run-time library and a set of header files that define system objects. The library is called `libpresto.a`. All of the necessary header files can be pulled in by including only the file `presto.h`. These files should probably be placed in `/usr/local/lib/libpresto.a` and `/usr/include/presto`. In addition to the PRESTO library, Sequent programmers will also need to include Sequent's parallel programming library dealing with the shared-memory structures on that machine. Eventually, PRESTO will provide its own memory management, eliminating the need for the Sequent library.

PRESTO relies heavily on inline expansions for simple, but frequently called functions (such as lock acquisition). These functions are defined within the PRESTO header files and are subject to change. Without inline functions, a change in the libraries only requires that user programs be relinked. In-line functions require that they be recompiled as well. For this reason, programmers are encouraged to specify the complete header dependencies within their makefiles. The following is a sample PRESTO makefile that takes care of creating these dependencies.

---

[1] The appropriate diffs to *cfront* are included in the appendix.

```
LIBS     =      /usr/local/lib/libpresto.a  -lpps
MAKEFILE =      Makefile
CC       =      CC
CFLAGS   =      -g
LDFLAGS  =      -Y             # if Sequent

SRCS     =      qs.c qsmain.c
OBJS     =      qs.o qsmain.o
PROG     =      qs

$(PROG):        $(OBJS) $(LIBS)
                $(CC) $(CFLAGS) $(LDFLAGS) $(OBJS) $(LIBS) -o $(PROG)
#
# Construct header dependencies
#
depend:
        $(CC) -M  $(SRCS)  |    sort | uniq > makedep
        cp $(MAKEFILE) $(MAKEFILE).sav
        sed -n '1,/^# DO NOT DELETE THIS LINE/p' $(MAKEFILE).sav >$(MAKEFILE)
        echo '# stuff after here goes away' >>$(MAKEFILE)
        cat makedep >> $(MAKEFILE)
        echo '# DEPENDENCIES MUST END AT END OF FILE' >> $(MAKEFILE)
        echo '# IF YOU PUT STUFF HERE IT WILL GO AWAY' >> $(MAKEFILE)
# DO NOT DELETE THIS LINE
# stuff after here goes away
```

The rules for making `depend` will update the makefile to properly reflect all header dependencies. It is important to keep these up-to-date. The only other interesting point about the makefile is the definition of `LDFLAGS` for the Sequent. The `-Y` will force static objects to be shared, *with the exception* of static class members. These need to be explicitly declared as shared.

```
class Any       {
        static shared_t int x;
};
```

2.4. Writing PRESTO Programs

A PRESTO program is essentially a C++ program having more than one thread of execution. A thread represents a virtual processor. There may be many more threads than physical processors when executing a PRESTO program. It is best to think of a thread as always executing in the context of one object or another. When an object synchronously invokes another object, the invoking object's thread is *borrowed* by the other, wherein it executes (possibly passing on to other objects) until it eventually returns. For consistency, global functions can be considered member functions of some single anonymous global object.

All PRESTO programs have three phases: initialization, execution and termination. These phases are defined in terms of member functions on the class `Main`, which is used to transform a program from a single-threaded UNIX process into a multi-threaded PRESTO program.

```
class Main      {
        int numprocessors;              // # scheduling processors
        int nummainthreads;             // # threads in Main::main()
        int mainstacksizes;             // how big each stack
        int quantum;                    // scheduling quantum
        int argc;                       // argc from main
        char **argv;                    // argv from main
        char **envp;                    // envp from main
public:
        Main(int ac, char **av, char **ep);
        ~Main();
        int init();             // user provides  any or all of
        int main();             //      init, main, done...
        int done();             //
};
```

The programmer provides the implementation for the functions `init()`, `main()` and `done()`, much as the UNIX programmer provides the implementation for the subprogram `main()` (which should not be provided in a PRESTO program). The initializing `Main::init()` is called in the context of a single-threaded UNIX program by the PRESTO run-time system (PRTS). In it, the programmer can specify PRTS parameters or create alternative system objects. `Main::init()` should return non-zero on failure, which will be passed through to `::exit()`. If the initialization does not fail, `Main::nummainthreads` will begin executing in `Main::main()`, running on top `Main::numprocessors` processors. If not set by `Main::init()`, the system will default to having one thread on one processor. For sanity's sake, `Main::numprocessors` is limited to one fewer than the number of physical processors on the machine. When the system runs out of runnable threads, it returns to a single-threaded UNIX mode and invokes `Main::done()`. The programmer can use this function to perform any extra cleanup that might be necessary. The function's return value is passed on to `::exit()`. The PRTS provides default implementations for the three functions `Main::init()`, `Main::main()` and `Main::done()`. These implementations do nothing, so the programmer who needs nothing can use them.

2.5. Basic Objects

The two main PRESTO classes for dealing with threads and synchronization are derived from the very basic object `Object`.

```
// objects.h
class Object    {
        int     o_type;                 // object type
        char    *o_name;                // object name
        Object  *o_next;                // linked list next field
public:
        Object(int type, char* name, Object* next = 0)
        int type()
        virtual void error(char* s);    // all objects handle their own errors
        char* name()
};
```

An instance of class Object has a name, type, an error handler, and the ability to "live" in an object queues (such as class Oqueue). PRESTO defines several object types in the file `objects.h`. Users can define their own objects having type values greater than `OBJ_END`. The main reason for the type is to allow error checking when dealing with objects in different capacities. The error handler provides a convenient way to lay the blame on an object when something goes wrong. The object can deal with the error in an appropriate manner. Most PRESTO objects abort the program when an error occurs – simple, but effective.

2.6. Threads

Although a thread represents the computational power of a virtual processor, a thread itself is represented by a normal C++ object. Creating and starting threads within an object's member functions are the most common operations.

```
//
// Abridged Thread interface.  FILE threads.h
//
class Thread : public Object    {
        Thread(char* name, int tid = 0, long ssiz = DEFSTACKSIZ, int musthavestack = 0);
        ~Thread();
        int     start(Objany obj, PFany pf, ...);
};
```

A thread has, among other things, a name, a thread id, and a stack. Creating a new thread is done with the standard `new` operator.

```
Thread *t = new Thread("billy", 100, 10*ONEK);
```

will create a new thread by the name "billy" with an id of 100 and a stack size of 10k. Thread names and thread id's are currently not used for anything in the PRTS other than to identify an offending thread when an error occurs. These values can be obtained using the operations `char *name()` and `int tid()`.

```
cout << "t's name is " << t->name() << " and id is " << t->tid();
```

The minimum stack size is 1024 bytes, the largest is 1 megabyte, and the default is 16k. For simplicity, stack sizes should be specified in increments of `ONEK` (or left to the default). The current implementation does no checking for stack overflow, although it probably should. The last argument to the thread constructor should only be given (as non zero) when a complete thread must be constructed. If a complete thread is not needed, the system may perform certain storage optimizations in order to minimize the amount of new shared data that must be created. These optimizations are discussed in detail in a later section.

A newly created thread is essentially a passive data object. That is, it has everything a thread needs *except* something to do. The operation *start()* enqueues a thread to begin executing within the member function of some object.

```
Thread *t = new Thread("billy");                // defaults are sufficient
Matrix *m = new Matrix;
Matrix *n = new Matrix;
t->start((Objany)m, (PFany)m->multiply, n);
```

is equivalent to the asynchronous invocation `m->multiply(n);`. `Start()` returns immediately, after having scheduled `t` for execution. A thread may only be started once or an error will occur. If the first argument to `start()` is NULL, the thread will be started in the global function named in the second parameter.

```
{
        extern int write(int fd, char* buf, int len);
        Thread *t = new Thread("writer");
#define HW      "Hello World"
        t->start(0, (PFAny)write, fileno(stdout), HW, sizeof(HW));
}
```

Although C++ has a reasonably strict typing system, `start()` is essentially untyped since it can be used to start a thread within any object's member function having any number of any type of parameters. The implementation of `start()` blindly copies its arguments onto the thread's stack without checking their types. Caveat Programmer.

The C++ compiler has its own ideas about what it means to take the address of a member function as is done with the call to `start()`. Newer versions of the compiler support the concept of a pointer to a member function, but not in the type-free way needed by PRESTO. These versions may emit the warning

```
"matrix.c", line 23: warning: address of bound function
(try Matrix ::* pointer type and &Matrix ::wait  address)
```

The warning can be ignored since the PRTS ensures that the operation is invoked on the correct object. It's also a good idea to always coerce the first two arguments of `start()` to their generic types (`Objany` and `PFany`) to avoid compiler error type-checking messages that can sometimes arise.

The asynchronous nature of `start()` affects the styles of parameter passing that may be used in a started function. Default and reference parameters will not work properly, although virtual and inline functions work fine. Overloaded functions will also not work, since the compiler can not discern the types of the arguments to the function (or its return value) at compile time.

Disallowing reference parameters implies that all parameters (including pointers) are passed by value. Consequently, the programmer should be careful that pointers reference objects residing in the global address space (static or created by `new`) and *do not* point to objects on the stack of the thread that is starting the new thread. Failure to abide by this may result in the passed pointer referencing garbage (consider the stack's behavior on an asynchronous invocation). Each of the following `start()` statements is erroneous.

```
class Foo        {
public:
        int use_pointer(int *x);
        int use_reference(int &x);
        int use_default(int x = -123);
        int use_overload(int x);
        int use_overload(float x);
};
....
{
        Foo    *f = new Foo;
        Thread *t = new Thread("timmy");
        t->start((Objany)f, (PFany)f->use_any, &x);             // pointer to stack variable
/*or*/  t->start((Objany)f, (PFany)f->use_reference, x);        // reference parameter
/*or*/  t->start((Objany)f, (PFany)f->use_default);             // default parameter
/*or*/  t->start((Objany)f, (PFany)f->use_overload,12);         // overloaded function
/*or*/  t->start((Objany)f, (PFany)f->use_overload, 8.5);       // overloaded function
}
```

A thread begins executing on its own stack. The variable `thisthread` always refers to the thread referencing it. For example, an object can query the name of the thread by which it is being animated.

```
cout << thisthread->name() << " - id == " << thisthread->tid();
```

When a thread returns from the function in which it was started, it will be reclaimed by the system. The programmer should be careful when referencing a thread after it has been started (garbage collection may cause the thread to become something different).

It is possible to wait on a thread's completion by expressing interest in the started function's return value after the thread has been created, but before it has been started.

```
t->willjoin();                              // preserve the return value
t->start(anyobject, anyfunction, args); // go
...                                       // stuff here
Objany tval = t->join();                    // wait here
```

`Objany` is a typedef for `void*` and can be used to name any four byte value. A thread may be joined by only one other. If a thread terminates, but has been marked as "joinable" with `willjoin()`, the thread will not be garbage collected until another thread joins with it. A thread may prematurely terminate itself[2] with

```
thisthread->terminate(obj)
```

where `obj` is an optional (default NULL) parameter of type `Objany`. If `thisthread` has been marked as joinable, the argument to `terminate()` is returned to the joining thread. The call to `terminate()` never returns.

An object may start a thread within itself, as well as within other objects.

```
t = new Thread("self");
t->start((OBjany)this, (PFany)this->func, arg1, arg2);
```

Among other things, this allows objects to animate themselves. In particular, a self-threading object can take advantage of static construction to form itself into an object that is *always* executing. The following example demonstrates this.

---

[2] Threads may terminate themselves only. There is no support (yet) for terminating other threads. Stay tuned.

```
#include <stream.h>
#include "presto.h"
class Busy      {
        char *n;
public:
        Busy(char *name, int count)
                { n = name;
                  Thread *t = new Thread(n);
                  t->start((Objany)this, (PFany)this->wait, count*1000);
                }
        int wait(int delay);
        ~Busy()
                {  cout << "BYE BYE from " << n; }
};
int
Busy::wait(int delay)
{
        while (delay-- > 0)
                if (delay % 1000 == 0)
                        cout << n << ":" << delay;
}
Main::init()
{
        numprocessors = 3;              // run on three processors
        return 0;
}

// STATIC CONSTRUCTORS
shared_t Busy   busyone("jim", 25 );
shared_t Busy   busytwo("fred", 40 );
shared_t Busy   busythree("isaac", 80 );
shared_t Busy   busyfour("bill", 15 );
```

The key point in this example is that it is not necessary to even provide an implementation for `Main::main()`. The declarations of the statics `busyone` through `busyfour` will cause the constructor for `Busy` to be invoked even before the PRTS begins.

There are some limitations to what can be done inside constructors that are called during static initialization. The order in which static constructors is called is not specified by the C++ language, so its possible for one static constructor to reference another static object that has not yet been constructed. This will likely cause a segmentation fault. In addition, since the PRTS may not yet be initialized during a static constructor's execution, certain operations are not guaranteed to work (or guaranteed not to work). Any operation that involves the scheduler (such as waiting on a condition variable) will cause a segmentation fault. The result of operations on `thisthread` are also unsafe. Objects may query a global variable to determine the current state of the system,

```
// presto.h
extern shared_t int prestoState;
#define STATIC_CTOR          0       /* running static constructors  */
#define MAIN_INIT            1       /* inside Main::init()          */
#define MAIN_MAIN            2       /* inside Main::main()          */
#define MAIN_DONE            3       /* inside Main::done()          */
#define STATIC_DTOR          4       /* inside static destructors    */
```

or use the macro `MULTITHREADED()` to determine the legitimacy of certain operations.

2.6.1. Forking

Creating and starting a thread can be combined into a single `fork` operation applied to the current thread `thisthread`. For example,

```
Thread* t = thisthread->fork(NOJOIN, (Objany)this, (PFany)this->wait, count*1000);
```

is equivalent to the code in the last example of a `Busy` constructor. If the first argument to `fork` is `WILLJOIN` then the return value can be used to join with the forked thread. It is not possible to fork on a thread other than `thisthread`.

2.7. Preemption

By setting the `Main::quantum` variable in `Main::init()`, PRESTO programs can take advantage of a preemptive scheduler. A preemptive scheduler imposes slightly more overhead, but can provide a more realistic mapping from virtual to physical processor. The quantum is specified in milliseconds. A quantum of zero implies no preemption. As the quantum decreases, program throughput may also decrease due to the overhead of dealing with many asynchronous scheduling interrupts. A quantum of 500 ms is reasonable, 100 is excessive, and 10 (the minimum) is absurd.

A thread may mark itself as non-preemptable,

```
thisthread->nonpreemptable();
```

or preemptable,

```
thisthread->preemptable();
```

to insulate itself from the preemptive scheduler.

2.8. Synchronization

Concurrent threads must be able to synchronize their actions. To allow this, PRESTO provides two basic types of synchronization: non-relinquishing and relinquishing.

2.8.1. Spinlocks

Non-relinquishing synchronization relies on spinlocks to force a thread to busywait on an event (generally, the release of a critical section of code). Spinlocks can be constructed, locked, unlocked, and conditionally locked. The following code fragment demonstrates how a spinlock might be used to control access to a shared critical region.

```
class MultiThreadedObject       {
        Spinlock        *sl;
public:
        MultiThreadObject()
                { sl = new Spinlock;}
        ~MultiThreadObject()
                { delete sl; }
        void entryPoint();
};

void MultiThreadedObject::entryPoint()
{
        sl->lock();
                // critical section
        sl->unlock();

        if (sl->checklock())    {       // return zero lock acquired
                // critical section
                sl->unlock();
        } else
                // can not acquire lock.  Take other action
}
```

When a thread holds a spinlock, that thread is not preemptable. A thread spinning, waiting for a lock, is preemptable though.[3]

---

[3] The current Sequent implementation does not consider the acquisition of a hardware lock and the marking of a thread as non-preemptable as an atomic event. To ensure correctness, a thread is marked as non-preemptable when it tries to acquire the hardware lock, rather than waiting until the lock has actually been acquired. Thus, a thread spinning on a lock will not be preempted. To do otherwise would introduce the possibility that a thread, having acquired a lock but not yet marked as non-preemptable, could be preempted. This can quickly lead to a deadlock situation. This problem will be fixed on the Symmetry.

2.8.2. Relinquishing Synchronization

When the expected waiting time for entering a critical section is high, spinlocks make very ineffective use of the underlying processors. Generally, it is more appropriate to relinquish the processor and reschedule the waiting thread when it would be allowed to enter the critical section. The three operations on threads that permit this are

```
// FILE: threads.h
class Thread ...        {
        ...
        void swtch();
        void sleep(SynchroObject* so);
        void wakeup(SynchroObject* so);
        ...
};
```

The first invokes the thread's lowest level scheduling primitive, relinquishing the processor to a system scheduling thread. The scheduling thread then runs any other ready thread that can be found. A thread can only sleep or switch out on itself. Trying to do otherwise raises an error on the thread.

Typically, threads don't just switch out. They go to sleep waiting for some event to occur, at which point they are "woken up". The classes describing these events are inherited from the general class `SynchroObject`.

```
// FILE: synchro.h
class SynchroObject : public Object {
        Spinlock        *so_lock;
        ThreadQUnlocked *so_waiting;
public:
        SynchroObject(int t, char *name);
        ~SynchroObject();
        void remember(Thread* t);       // remember thread blocking
        Thread* recall();               // get blocked thread
        virtual void error(char* s);
        inline void lock();             // spin on access to object
        inline void unlock();
        ThreadQUnlocked *waitingQueue();// return waiting queue
};
```

The base class `Object` allows SynchroObjects to be maintained in queues, have names, etc. (see objects.h). SynchroObjects use a simple spinlock to control access to the object's data structures. To see why this is needed, consider two threads trying to acquire a monitor at the same time. The `so_waiting` queue keeps track of all threads waiting on a given SynchroObject.

2.8.3. Useful Synchronization Objects

SynchroObjects have no usage semantics. That is, a thread can not wait on a SynchroObject directly. They exist as a base class for other relinquishing primitives that do have semantics. For example, a simple relinquishing lock is defined in terms of the basic SynchroObject.

```
// synchro.h
class Lock : public SynchroObject       {
        Thread  *lo_owner;              // who owns the lock
        void    lock2();                // looping wait
public:
        Lock(char *name);
        ~Lock();
        inline void lock();             // acquire
        inline void unlock();           // release
        void error(char *s);
        Thread *owner()
                { return  lo_owner; }
};
```

A `Lock` is nothing more than a SynchroObject that has an owner (`lo_owner`) and some operations (`lock()` and `unlock()`).

Performance is the motivation for in-lining the lock function. On the Sequent, the speedup can be as much as one-third.[4] On machines with better support for atomic locking (such as the Symmetry), the improvement due to inline expansions will be less. Eventually, the more complicated inline functions will not be inlined.

Another point relating to inline functions is their inability to describe loops. A looping implementation of `Lock::lock()` would be written as:

```
void Lock::lock()
{
        SynchroObject::lock();                  // acquire base spinlock
        while (lo_owner)        {
                // update thisthread's state to reflect new waiting status
                remember(thisthread);           // enqueue on SynchroObject
                SynchroObject::unlock();
                thisthread->sleep(this);        // go to sleep here
                SynchroObject::lock();          // reacquire base spinlock
        }
        lo_owner = thisthread;                  // it's mine!
        SynchroObject::unlock();
}
```

but the C++ compiler balks at the while loop. To solve this, the lock implementation is broken into two parts: a non-looping part which can be inlined, and a looping part that is called if the lock is already held. The rationale is that if the lock is already held, the overhead of an extra procedure call will be small compared to the waiting time of the lock. This trick is used in several places throughout the system.

```
inline
void Lock::lock()
{
        SynchroObject::lock();
        if (lo_owner == 0)      {
                lo_owner = thisthread;
                SynchroObject::unlock();
        } else
                lock2();
}
```

where `lock2()` simply implements the while loop in the first version of `lock()`.

A thread can put only itself to sleep, but any thread can cause another thread to wakeup.

```
inline
void Lock::unlock()
{
        SynchroObject::lock();
        lo_owner = 0;
        Thread *newowner = recall();    // find new lock owner
        SynchroObject::unlock();
        if (newowner)
                newowner->wakeup((SynchroObject*)this);
}
```

`Wakeup()` ensures that the thread being awoken is indeed sleeping, and then enqueues it for running. It is an error to wakeup a thread which is not sleeping.

In addition to simple relinquishing locks, PRESTO also offers monitors and condition variables. Together, these implement the Mesa synchronization semantics [4]. There is no compiler support for these objects, so it is not possible to declare an entire object or function as "monitored." This fact must be included in the definition. For each function that should be guarded by a monitor, it is necessary to include the code marking the monitor's entry and exit at the appropriate points. The following code replaces the spinlock in the MultiThreadedObject class above with a monitor and adds a condition variable.

---

[4] A procedure call takes about 15μsecs. Acquiring and releasing the SynchroObject's spinlock takes about 30μsecs. An synchronization object that does *nothing* could take 45μsecs without inline expansion, but only 30 otherwise.

```
class MultiThreadedObject       {
        Monitor         *mto_mon;
        Condition       *mto_cond;
        int             mto_ok;
public:
        MultiThreadObject()
                { mto_mon = new Monitor("mto_monitor");
                  mto_cond = new Condition(m, "mto_condition");
                  mto_ok = 0;
                }
        ~MultiThreadObject()
                { delete mto_mon; delete mto_cond; }
        void entryPoint();
};

void MultiThreadedObject::entryPoint()
{
        mto_mon->entry();
                // critical section
                while (! mto_ok )
                        mto_cond->wait();
                // more critical section
                if (mto_ok)
                        mto_cond->signal();
        mto_mon->exit();
}
```

A condition variable must be bound to a monitor. It is an error for a thread to operate on a condition variable if that thread does not currently hold the associated monitor. Similarly, a thread may not exit a monitor (or release a lock) that it does not hold. These run-time restrictions are necessary since the compiler does not enforce special scoping checks for condition variables. There is an advantage to this though: condition variables and monitors may be passed as arguments to functions.

Monitors can be created (new), destroyed (delete), entered, exited, queried for their owner, printed, and laid blame upon.

```
class Monitor : public Lock             {
public:
        Monitor(char* name=0);
        ~Monitor();
        inline void entry()
                { Lock::lock(); }
        inline void exit()
                { Lock::unlock(); }
        Thread *owner()
                { return Lock::owner(); }
        virtual void print(ostream& = cout);
        // error function derived from Lock
};
```

Having to explicitly enter and exit each monitored critical section can be inconvenient and error-prone. To address this, a syntactically sugared construct exists that allows a monitor to control access to a C++ bracketed scope. When flow of control leaves that scope, the monitor is automatically released.

```
// FILE: synchro.h
class MONITOR   {
        Monitor *mo_mon;
public:
        MONITOR(Monitor *m)
                { mo_mon = m; m->entry(); }
        ~MONITOR()
                { mo_mon->exit(); }
};
```

Instances of class `MONITOR` should only be declared on the stack. Their only purpose is to take advantage of the fact that an object's destructor is invoked automatically when that object goes out of scope. This relieves the programmer of the responsibility of having to explicitly exit a monitor. The monitored member function in the previous example can be rewritten as:

```
void MultiThreadedObject::entryPoint()
{
MONITOR ENTRY(mto_mon);          // ENTRY is a nice sounding dummy var

        // critical section
        while (! mto_ok )
                mto_cond->wait();
        // more critical section
        if (mto_ok)
                mto_cond->signal();
}
```

The operations on a condition variable are create, destroy, wait, signal, broadcast and error.

```
// FILE: locks.h
class Condition : public SynchroObject  {
        Monitor*        co_monitor;             // bound monitor
public:
        Condition(char* name=0);        // unbound
        Condition(Monitor* boundmon);   // bound, no name
        Condition(Monitor& boundmon);
        Condition(Monitor* boundmon, char* name);
        Condition(Monitor& boundmon, char* name);
        Monitor* monitor()
                { return co_monitor; }
        int threadok()                          // is cond user legit
                { return ((!co_monitor) ||      // unbound is free4all
                         (thisthread == co_monitor->owner()));}
        ~Condition();
        void signal();                  // wakeup one
        void broadcast();               // wakeup all
        void wait();                    // wait for signal (could pickup old)
        virtual void print(ostream& = cout);
};
```

Deleting a monitor or condition variable on which other threads are waiting raises an error on that monitor or condition variable.

Because condition variables obey Mesa semantics, a signal or broadcast should only be considered as a *hint* that some condition stronger than the monitor invariant *had* been established. The condition may no longer hold. The general form for condition variables is

```
while (condition_not_true)      {
        condition->wait();              // relinquish the monitor
};                                      // reacquire the monitor
```

2.9. Debugging PRESTO Programs

For the most part, debugging PRESTO programs is similar to debugging normal C++ programs. The main difference comes about in dealing with multiple threads. The present UNIX debuggers (adb, ddt, dbx) do not know how to deal with multiple threads. Consequently, inspecting and controlling multiple threads must be done manually. Although not difficult, it does require a bit of finesse. The situation is further complicated by the fact that the existing UNIX debuggers understand only C, but not C++.[5] Nevertheless, debugging C++ programs with a C debugger is not difficult and requires only a little experience. This section first describes the general concepts needed to debug normal C++ programs, and then explains how the use the dbx debugger with multiple threads.

---

[5] The GNU debugger gdb works with the GNU version of C++, g++. The GNU software is not yet stable though, so it is too early to tell if it is worth adopting in preference to the AT&T "official" version.

2.9.1. Rules For Debugging C++ Programs

1 The C++ compiler is a true compiler that translates C++ code into C code, much as the C compiler translate C into assembly language. By using the +i option to the compiler, one can view (and hence debug) the C code directly. Without the +i option, the translator inserts the necessary *#line* directives allowing the debugger to map backwards from the C code to the C++ code. It is generally easier to debug the C++ code, but sometimes one can only see what is really happening by looking at the C code. This is analogous to examining the assembler output when the compiler doesn't seem to be doing what it's been asked. The library `libpresto_c.a` contains PRESTO routines compiled with the +i option.

2 There is a very regular translation from C++ names to the C names. A member `M` of a C++ class `Q` becomes a C structure element `_Q_M`. Example:

```
case Foo        {
        int x;
public:
        Foo();
};

f = new Foo;
```

to reference f->x, one would use

```
f->_Foo_x.
```

The same rule holds for member functions.

3 Constructors and destructors are renamed to `__ctor`, `__dtor`. Example:

```
f = new Foo;
delete f;
```

translates into calls to `_Foo__ctor` and `_Foo__dtor`.

4 Static class members become global variables prepended by the name of the class. Example:

```
class Foo       {
        static shared_t int ff;
};
```

translates into the declaration

```
shared_t int _Foo_ff;
```

5 Stack variables (including function parameters) are prepended with an `_auX_`, where `X` is an integer equal to the nesting depth of the variable's declaration. Function parameters are at depth 0. Example:

```
int somefunc(int x)
{
        int y;
        y = 10;
        { int z = 100; }
}
```

translates into

```
int somefunc(au0_x)
        int au0_x;
{
        int au1_y;
        _au1_y = 10;
        { _au2_z = 100; }
}
```

To see all the variables in use, use the dbx command "dump."

6 Every C++ class member function receives a hidden first argument `_au0_this` which is a pointer to the object for which the member function is defined. Example:

```
class Foo       {
public:
        int bar(int x);
};
int
Foo::bar(int x)
{
        return x + 1;
}
```

translates into

```
int
_Foo_bar(_au0_this, _au0_x)
        struct Foo *_au0_this;
        int _au0_x;
{
        return _au0_x + 1;
}
```

To dump a whole struct, use `print *_au0_this` and the element names will be displayed in their the C form.

7 Virtual functions are looked up in a table before calling. Each class Q having virtual functions has a hidden element _Q__vptr. During construction of an object of class Q, _Q__vptr is initialized to point to a global table Q__vtbl. This table contains the addresses of the virtual functions. When calling the *i*th lexically defined virtual function for a class, the *i*th entry in the table is used to find the actual function to call.

8 Overloaded functions are qualified by their argument type. Example:

```
overload int foo(int x);
overload int foo(double y, int x);
```

produces declarations for functions

```
int foo();              // first function is named normally
int fooFD_I();          // function taking a double and an int
```

The pattern is straightforward and works the same way for member functions.

9 Member elements of a base class B are textually included in any classes derived from B. Example:

```
class Base      {
        int x;
};
class Derived : Base    {
        int y;
};

int x = sizeof(Derived);
```

will assign 8 to `x`.

2.9.2. Debugging Multiple Threads

A thread can loosely be defined as the combination of a program count (pc) and a stack pointer (sp). At every point in the thread's execution, the pc has a value, and the sp has a current value. The pc's value corresponds to the next instruction to be executed by the thread, and the sp points to the bottom of the stack of call-frames representing the thread's current execution state. The debugger understands only single-threaded programs and always responds to queries using the current execution state; that represented by `thisthread`.

Examining a thread that is not running is possible by changing the debugger's idea about what is the current state. For example, suppose its necessary to examine all threads blocked on a condition variable `ondition *cond`. The condition variable is an instance of a SynchroObject, and so maintains a waiting queue. The dbx command

```
print *cond
```

will display the address of `cond->_SynchroObject_so_waiting`. The queue is an instance of class Oqueue, so it has a head, and each element in the queue is an instance of class Object. There are two ways to examine the queue. The first is to plod through the list manually from within dbx, looking at each objects, and then the next. The other (simpler) way is to call on an object's print routines directly, passing it the "hidden" first argument `this`.

```
call _Condition_print(cond->_SynchroObject_so_waiting, &cout)
```

The print function will display the state of the condition variable, and then invoke `print` on the waiting queue, which will invoke `print` on each element in the waiting queue. All PRESTO objects can display themselves. If the information displayed when an object is printed is not sufficient to answer a debugging query, then it is necessary to poke at the object directly.

It is possible to view the stack of a blocked thread by changing the debugger's notion of the current frame to the bottom-most frame of the blocked thread. For example, if `_aul_t` is a pointer to a thread in a PRESTO program, that thread's stack can be examined by doing:

```
dbx>set oldfp = $fp
dbx>assign $fp = _aul_t->_t_fp
dbx>where
dbx>dump
dbx>assign $fp = oldfp
```

This simply stores the current frame pointer, installs a new one, dumps the stack, and then restores the old frame pointer. A thread's `t_fp` field *always* refers back to the frame that should be loaded when the thread is next involved in a context switch. So, if the thread is currently active, `t_fp` is the frame pointer for a switch-back scheduling thread. If the thread is blocked, `t_fp` points to the blocked thread's last active frame.

It is not generally possible to single-step a blocked thread, since the thread isn't runnable. It *should* be possible to force a context switch to any other thread waiting on the ready queue, but a reasonable interface allowing this does not yet exist.

3. Customizing PRESTO

PRESTO can be customized to create any one of a number of parallel programming environments. The loose structure of the system's components make this possible. There are five fundamental PRESTO objects: the scheduler, the processor, the thread, the spinlock and the synchronization object. Figure I illustrates these objects in terms of their inter-object relationships. From these five, the system supplies a basic parallel programming environment that includes

- a preemptive scheduler,
- the ability to create new threads of control,
- busywaiting synchronization based on hardware atomic locks, and
- primitives allowing a thread to deschedule itself and be rescheduled by another.

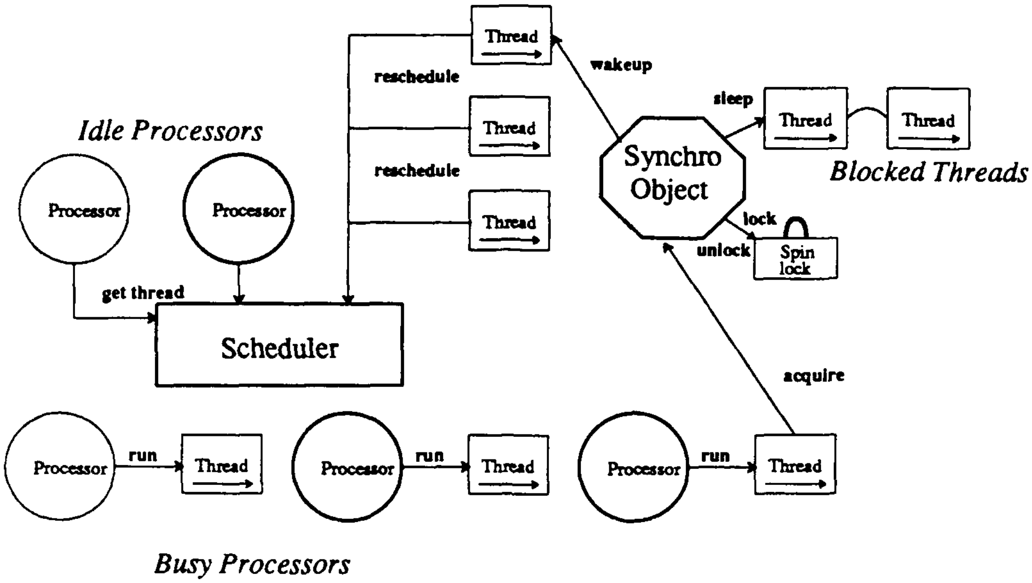

Idle Processors
Processor
Processor
get thread
Scheduler
reschedule
reschedule
Thread
Thread
Thread
wakeup
sleep
Synchro Object
lock
unlock
Spin lock
Thread
Thread
Blocked Threads
acquire
Processor
run
Thread
Processor
run
Thread
Processor
run
Thread
Busy Processors

Figure 1 – Presto Components

Threads can be created, destroyed, put to sleep and awoken. A thread is put to sleep on a synchronization object, which consists of a queue, a spinlock, and whatever other state is needed to implement the synchronization object's semantics. The synchronization objects provided by PRESTO have no semantics in the sense of *P* and *V* [3] or *signal* and *wait*. The spinlock is needed to guard the critical sections that describe a given synchronization object. Deciding when to block and enqueue a thread that tries to pass through a synchronization object is not PRESTO's responsibility. Synchronization *policy* belongs to the parallel programming model, not to PRESTO.

The scheduler maintains a pool of runnable threads. Threads enter the pool when they become ready, and processors empty the pool when they become idle. The processor is an inherently active object, while the scheduler and threads are inherently passive. The main body of the processor object supplied by PRESTO does

```
forever do
    ask scheduler for the next ready thread
    if a ready thread is available
        ask the ready thread to run
    else if there will never be a ready thread again
        quit
```

When a processor asks a thread to run, the thread becomes active using the thread of the processor. Once active, the thread is able to execute within any other object. The thread runs until it is preempted, goes to sleep on a synchroniztion object, or terminates. After any of these actions, the processor object reactivates and continues looking for ready threads. If a processor idles, finding nothing to do, and all other processors are idle, the system halts.

The basic technique for modifying PRESTO objects is

1. Define a new PRESTO object derived from the basic object to be modified.
2. Create a new instance of this object.
3. Inform the PRTS that this new instance should be used instead of the default provided by PRESTO.

For example, to create a new scheduler, one would define a new scheduling object

```
class NewScheduler: public Scheduler {
        // differences described here
};
```

An instance of this object can then be created and bound to the name `sched`.

```
NewScheduler *sched = new Scheduler(/*Constructor arguments here*/);
```

The PRTS will instantiate its own scheduler after `Main::init()` returns and if the PRESTO name `sched` is unbound. Rebinding `sched` in routines other than `Main::init()` will only work properly if the new scheduler knows how to take control from the existing one.

Part of the PRESTO scheduler's job is to create process objects (class Process) and scheduling threads to animate those process object. Rather than invoking the `new` operator on those classes, the scheduler uses the prototypical `thisthread` and `thisproc` to obtain new instances.

```
Thread* t = thisthread->newthread(/*Constructor arguments here*/);
Process* p = thisproc->newprocess(/*Constructor arguments here*/);
```

The operators `newthread()` and `newprocess()` serve as virtual constructors for the process and thread class. Assignment to `thisthread` and `thisproc` in `Main::init()` forces the scheduler to use the virtual constructors for the type of class assigned. These are typically defined as:

```
Thread*
SomeKindOfThread::newthread(char* name, int tid, int stacksiz, int other)
{
        return (Thread*)new SomeKindOfThread(name, tid, stacksiz, other);
}
```

where `SomeKindOfThread` is derived from the basic class `Thread`.

Appendix A – C++ Diffs For The Sequent

It is necessary to add the keywords `private_t` and `shared_t` to the version of `cfront` running on the sequent. The following diffs should be applied to the AT&T version of C++ (Old) to generate a Sequent compatible version (New). These diffs are also included on the distribution tape in the file `C++.diffs.sequent`.

```
diff Old/cfront.h New/cfront.h
339a340,343
> #ifdef ns32000
>       bit     b_private_t;
>       bit     b_shared_t;
> #endif
diff Old/lex.c New/lex.c
192a193,197
>
> #ifdef ns32000
>       new_key("shared_t", SHARED_T, TYPE);
>       new_key("private_t", PRIVATE_T, TYPE);
> #endif
diff Old/norm.c New/norm.c
129a130,133
> #ifdef ns32000
>       case SHARED_T:  b_shared_t = 1; break;
>       case PRIVATE_T: b_private_t = 1; break;
> #endif ns32000
238a243,249
>
> #ifdef ns32000
>       /* not valid to try to save space on shared and private data types */
>       if (b_private_t || b_shared_t)
>               return this;
> #endif
>
diff Old/norm2.c New/norm2.c
149a150,153
> #ifdef ns32000
>       case PRIVATE_T: b_private_t = 1; break;
>       case SHARED_T:  b_shared_t = 1; break;
> #endif
diff Old/print.c New/print.c
37a38,45
>
> #ifdef ns32000
>       if (t == PRIVATE_T)
>               putstring("private");
>       else if (t == SHARED_T)
>               putstring("shared");
>       else
> #endif
51c59,60
<
---
>
>
93a103,107
> #ifdef ns32000
>       if (b->b_shared_t) puttok(SHARED_T);
>       if (b->b_private_t) puttok(PRIVATE_T);
> #endif ns32000
>
701a716,717
>
>
706a723
>
708a726
>
```

References

1. Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. Technical Report 87-09-01, Department of Computer Science, University of Washington, September 1987. Submitted for publication.
2. Brian N. Bershad, Edward D. Lazowska, Henry M. Levy, and David B. Wagner. An Open Environment for Building Parallel Programming Systems. Technical Report 88-01-03, Department of Computer Science, University of Washington, January 1988. Submitted for publication.
3. E.W. Dijkstra. The Structure of the 'THE' Multiprogramming System. *Communications of the ACM 11*,5 (May 1968), pp. 341-346.
4. B.W. Lampson and D.D. Redell. Experiences with Processes and Monitors in Mesa. *Communications of the ACM 23*,2 (February 1980), pp. 104-117.

An Open Environment for Building Parallel Programming Systems

Brian N. Bershad, Edward D. Lazowska,
Henry M. Levy and David B. Wagner
Department of Computer Science
University of Washington

Technical Report 88-01-03
January 1988

An Open Environment for Building Parallel Programming Systems

Brian N. Bershad
Edward D. Lazowska
Henry M. Levy
David B. Wagner

Department of Computer Science
University of Washington
Seattle, WA 98195

January 1988

ABSTRACT

PRESTO is a set of tools for building parallel programming systems on shared-memory multiprocessors. PRESTO's goal is to provide a framework within which one can easily build efficient support for any of a wide variety of "models" of parallel programming. PRESTO is designed for easy modification and extension, not only at the level of the primitives and structures made available for the application programmer's use, but also at the level of the run-time kernel that supports parallel applications. PRESTO is implemented in the object-oriented language C++ on a Sequent Balance 21000 and has been used in a number of applications that are described in this paper.

1. Introduction

Most parallel programming systems present themselves in terms of a fixed set of primitives (e.g., send a message, receive a message, acquire a monitor) running on top of a closed run-time kernel. The primitives together with the kernel define a "model" of parallel programming that, while pleasing to the implementor, may not always be satisfactory to the application programmer. Should incompatibility arise (e.g., due to the demands of a particular application), the programmer must either find another system, or build one, or attempt to conform to the parallel programming model supported by the available system. The first option may be impossible since there may not be another system that runs on the given hardware. The second option, while possible, is likely to be prohibitively expensive. This leaves only the third choice – shoehorning.

PRESTO addresses this dilemma. PRESTO is a set of tools for building parallel programming systems on shared-memory multiprocessors. PRESTO's goal is to provide a framework within which one can easily build efficient support for any of a wide variety of of parallel programming models. PRESTO has been used to emulate existing models, and to create new ones. Among these are a Mesa-like environment, one providing ACTOR-like futures, and one for writing parallel simulations. The ease with which support for new parallel programming paradigms can be built using PRESTO allows the programmer to choose for him or herself the model of parallel computation that is most appropriate to a given problem. (Of course, support for various common models is intended to be built once and then shared.) Figure 1 illustrates the relationship between PRESTO, the various parallel programming systems implemented in PRESTO (in heavy boxes), and the various applications implemented in each of these parallel programming systems (in dashed boxes). Each of these systems is discussed in a later section.

Our work is supported by the National Science Foundation (Grants No. CCR-8619663, CCR-8703049, and CCR-8700106), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, the USENIX Association, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

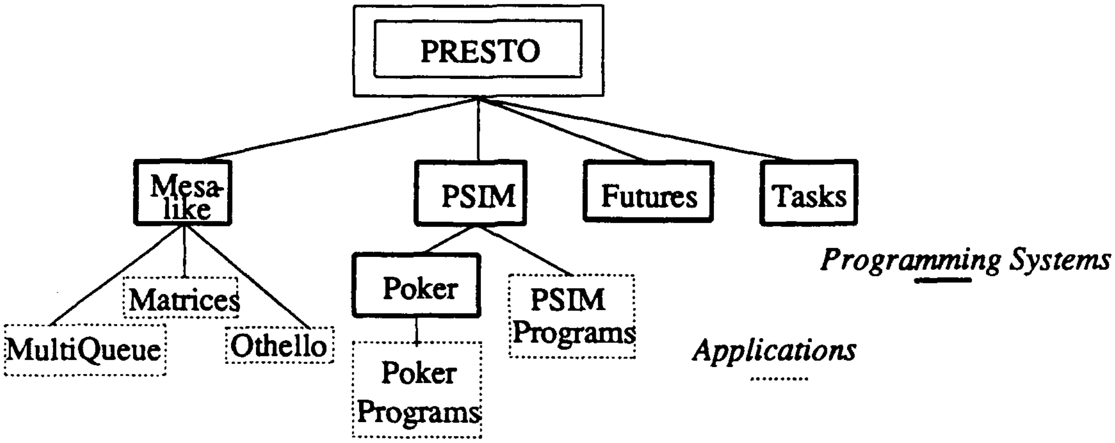


Figure 1 – PRESTO, Parallel Programming Systems, and Applications

PRESTO's "openness" – its ease of modification and extension – applies not only to the primitives and structures made available for the application programmer's use, but also to the lowest levels of the run-time kernel. Many implementors of new programming systems in PRESTO would not be concerned with the low-level details of a multiprocessor environment such as scheduling, preemption, and processor control: they would take what PRESTO gives them in these areas. However, if the characteristics of a new environment do require exceptional handling in any of these low-level areas, changes are possible through exactly the same mechanisms used to customize PRESTO at higher levels.

2. Achieving an Open System Through Object-Oriented Design

A parallel programming environment must deal with issues such as processor control, scheduling, concurrency, and synchronization. PRESTO encapsulates each of these issues inside a default structure having a fixed interface. Programming systems utilize these structures. Sometimes the basic structures serve the needs of a programming system, sometimes the structures need to be modified or extended in order to be useful, and sometimes the structures must serve as a base for other, higher-level structures. PRESTO allows this degree of customization by employing an *object-oriented* programming paradigm.

Objects are recognized as providing an effective means for structuring sequential software systems in terms of components and interfaces. An object has a name, private data, and a set of interfaces that allow other objects to view and manipulate it. The interfaces serve to contain and insulate an object's state, so that its own implementation is invisible to those who use it.

In terms of PRESTO's goals, the most important aspect of an object-oriented environment is the ability to redefine an object's behavior. As long as the object's interface remains unchanged, other objects in the system need not be informed of the changes. This property allows the designer to modify the behavior of system objects.

As well as being an ideal vehicle on which to structure an open system such as PRESTO, objects are also a useful abstraction for writing parallel programs. An object can maintain its own internal parallelism, *and* control any concurrency imposed upon it by other objects [20]. We use these points to argue that PRESTO-derived programming systems should present object-oriented structures to their (application) programmers. In most cases, these systems have done so, although their have been exceptions.[1]

---

[1] Specifically, the Poker simulation environment is programmed in C, a language that is definitely not object-oriented.

HYDRA [18] was the first system to adopt an object-oriented view to address the fact that the design of parallel systems is as much an art as a science. Both HYDRA and PRESTO are open systems in recognition of the fact that there is no "right" way to build a system for a parallel machine, that *any* system should have a clear separation between mechanism and policy, and that strict hierarchical layering of system components limits flexibility. Unfortunately, in a full-blown operating system such as HYDRA these principles must be balanced against real-world issues such as protection, fairness and reliability. Consequently, much of the openness may be compromised. For example, it is infeasible for an operating system to permit easy redefinition of the concepts of a processor, a lock, or even a thread. These are the most basic components of an operating system, and allowing users the freedom to change them could result in chaos. Indeed, there is no evidence in the literature suggesting that these types of objects were ever redefined in HYDRA. PRESTO runs on top of existing operating systems, and provides full flexibility in those areas that are critical to the construction of parallel applications.

3. The PRESTO System Structure

The most important aspect of PRESTO's design is its simplicity, both in concept and in implementation. The conceptual simplicity allows programmers to quickly grasp the functions that the system *does* provide, while the simplicity of the implementation allows them to introduce extensions without concern for subtle interactions between components.

There are five fundamental PRESTO objects: the scheduler, the processor, the thread, the spinlock, and the synchronization object. From these, the system supplies a basic parallel programming system that includes:

- a preemptive scheduler,
- the ability to create new threads of control,
- busy waiting synchronization based on hardware atomic locks, and
- primitives allowing a thread to deschedule itself and be rescheduled by another thread.

Threads are created, destroyed, put to sleep, and awakened. A thread is put to sleep by a synchronization object, which consists of a queue, a spinlock, and whatever other state is needed to implement the synchronization object's semantics. The spinlock guards the critical sections that describe a given synchronization object. The synchronization objects provided by PRESTO have no semantics in the sense of *P* and *V* [5] or *notify* and *wait* [14]. PRESTO interprets them merely as objects on which threads can be blocked, queued and resumed. The policies governing when to block are provided by more sophisticated synchronization objects derived from the basic ones provided by PRESTO.

The scheduler maintains a pool of runnable threads. Threads enter the pool when they become ready, and processors extract threads from the pool when they become idle. The main body of the processor object supplied by PRESTO does

```
forever do
    ask scheduler for the next ready thread
    if a ready thread is available
        request the ready thread to run
    else if there will never be a ready thread again
        quit
```

When a processor requests that a thread run, the thread becomes active, using the power of the requesting processor. Once active, the thread is able to execute within any other object. The thread runs until it is preempted, goes to sleep on a synchronization object, or terminates. After any of these actions, the processor object reactivates and continues looking for ready threads. If a processor idles, finding nothing to do, and all other processors are idle, the system halts. Figure 2 illustrates the main PRESTO components "in action," highlighting the states of threads as they progress through the system.

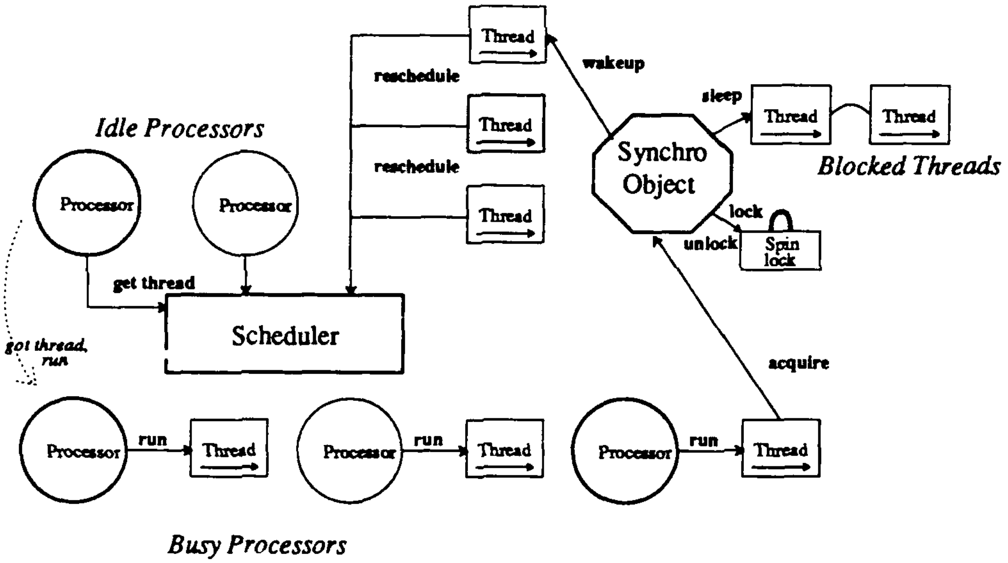


Figure 2 – PRESTO Components

4. Techniques For Customizing PRESTO

There are three basic methods for customizing the PRESTO environment: layered extension, differential extension, and lateral extension. The first is a property of all programming languages, the second of most object-oriented languages, and the third of open systems such as PRESTO.

Layering allows the programmer to build new primitives through the composition of existing ones. There are three problems with layering. First, a layered system's performance can quickly degrade as layers are added. Second, it may be difficult to express a new abstraction in terms of the existing ones. For example, consider the distinction between Hoare monitors [9] and Mesa monitors [14] . The stricter semantics of Hoare monitors makes them difficult to implement on a multiprocessor, and even more difficult (and costlier) if they must be implemented on top of less stringent Mesa monitors. Finally, layering is only useful for describing the behavior of something new; existing code cannot be affected.

Differential extension allows the programmer to exploit the hierarchical type system of an object-oriented programming language having inheritance. New classes may be differentially specified in terms of existing ones. This allows the programmer to construct classes similar to existing ones, while only specifying the changes in the classes' behavior. Hierarchical extensions can be combined with layering, so that new or modified operations invoke the primitive ones in the more basic class. Operations that are unchanged are executed directly.

Lateral extension makes it possible to change the behavior of an object dynamically (the other extensions are specified at compile-time) by affecting its relationships with other objects. For example, PRESTO provides a global scheduler object responsible for mapping runnable threads onto idle processors. The system's processor objects interact with the scheduler object. By replacing the system's scheduler object with a different one, the programmer can radically affect the behavior of the system. Lateral extensions such as this allow the programmer to install changes at any level in the system.

Layered, differential, and lateral extensions can be combined to achieve new results with a minimum of coding and effort. System components can be redefined differentially and installed laterally. When appropriate, the new version of an operation can be layered on top of existing ones. The programmer never changes the basic PRESTO classes, but instead derives new ones from them. Instances of these newer classes are created, and then bound to the names used by other objects so they can be referenced. For example, to force the scheduler to use priority scheduling instead of the default first-come-first-served, the following would be done:

derive a class ThreadPriorityHeap from the system class ThreadPool
define the operations get and put for a ThreadPriorityHeap
create a new instance of ThreadPriorityHeap
inform the scheduler to replace its current thread pool with the new ThreadPriorityHeap

When given a new thread pool, the scheduler moves all threads from the old pool into the new pool, allowing the scheduling discipline to be changed dynamically. For large parallel applications that compute in phases, this flexibility is important.

5. Exploiting PRESTO's Flexibility

To add concreteness to the preceding discussion, we will present three customizations of quite different styles that have been implemented using PRESTO's open architecture. The first customization involves a redefinition of one of the system's most fundamental objects, the thread, in order to gather data about parallel program performance. The second customization involves the construction of a number of higher-level general parallel programming environments, each providing its own set of primitives and model of programming. The third customization involves the construction of specialized environments for building parallel programs in narrow application domains; in particular, a PRESTO-based environment for writing parallel discrete-event simulations. The characteristics of these narrow-domain applications often require special handling at very low levels in order to obtain reasonable performance; the simulation environment provides its own scheduler for handling situations that are particular to the simulation world.

5.1. Building an Instrumented Execution Environment

Understanding how to improve a parallel application requires understanding its behavior. This, in turn, requires the ability to observe the run-time characteristics of an application (either in real time, or in "play-back" mode). Expecting the designers of a programming environment to include code for monitoring *all the right things* is unreasonable. Redefinition allows the programmer to use the base system components while collecting data about their behavior.

PRESTO objects have been instrumented to permit the collection of data during an application's execution. The instrumentation is implemented by creating, for each PRESTO system class, an instrumented version of that class. For example, once an application has been written, a programmer might need to know the percentage of time that a thread is blocked with respect to its total lifetime. This information is easily obtainable by deriving a new class *InstrumentedThread* from the basic PRESTO class *Thread*. An *InstrumentedThread* redefines the two thread operations *sleep* and *wakeup* so that they adjust a timer before invoking the real *sleep* and *wakeup* operations in the superclass class *Thread*.

System objects that create new instances of other system objects rely on a prototypical instance of the new object to direct the creation. Part of the scheduler's responsibility, for instance, is to create one thread per processor in the system. These threads busy themselves by looking for something to do, doing it, and then continuing looking. The scheduler does not create these new threads directly; instead, it asks its own thread (since the scheduler must be running, it must have a thread) to create a new instance of itself. By specifying the scheduler's prototypical thread, the programmer can control the behavior of all other system threads. Were PRESTO to assume that threads always looked the same, this would not be possible.

5.2. Building Various Parallel Programming Environments

A First Model

The first real applications built on top of PRESTO were implemented using a Mesa-like environment that can be included when a PRESTO program is compiled. The environment simply provides threads (Mesa *processes*) that can fork other threads, join on their results, and synchronize using the standard Mesa monitors and condition variables. In a sense, the environment is bland, but it *is* effective, and it was cheap to build. The entire implementation required only about 200 lines of code when built using PRESTO.

Despite its wide appeal, Mesa's model of parallel programming is not without flaws. Threads have special semantics apart from other program objects, and synchronization constraints must be made explicit by the programmer. Nonetheless, we have used this programming model for a wide range of applications that include matrix multiplication, sorting, analysis of multi-class queueing networks, and a parallel Othello program.

Implementing Higher-Level Abstractions

The primary goal of any parallel programming environment is to help make it easier to write and reason about parallel programs. To this end, there have been a large number of notable systems [1,6-8,10-13,19], each providing their designers' notions of the best possible environment for constructing parallel applications. Although each system is unique in its goals and implementation, each has had to address a similar set of key questions. By answering these questions differently, different models of parallel programming are realized. Rather than enumerating the features of each system and discussing its (potential) implementation in PRESTO, we instead touch on some of the key questions, discussing how they would be answered by PRESTO.

- *Should dynamic creation of threads be allowed?*

  Concurrent Euclid and CSP do not support the dynamic creation of new threads. The basic process structure is specified at compile time and never changes. Although thread creation in PRESTO is inherently dynamic, static objects containing their own thread(s) of execution can be declared at compile time, and the freedom to create new threads at run time can be denied.

- *Should objects and threads be disjoint or unified concepts?*

  Systems such as Smalltalk-80, Mesa, and Modula-2+ make a strong distinction between an object, which is inherently passive, and a thread (or process), which is used to animate objects. Threads may not even be first class objects. This dichotomy can be confusing to programmers, making it difficult to model problems and enforce protection among objects. On the other hand, systems such as ConcurrentSmalltalk and Act 1 unify the two notions. An object is a schedulable entity in these systems, and there is no conceptual gap between an object and a thread – one cannot exist without the other.

  PRESTO supports both models. Threads can be separate entities from the objects in which they execute, or threads and objects can be unified by defining classes that *thread* themselves upon instantiation. The PRESTO-derived class `Task` serves this function. Tasks execute autonomously, communicating with one another via messages. The basic structure of a task is

  ```
  forever do
      m ← receiveMessage
      decode m
      execute the operation requested in m resulting in a new message m'
      sendMessage m' to the sender of m
  ```

  A user's definition for an object derived from a `Task` has no main body. It just provides the operations that are invoked by the task's thread upon receiving a message from another task. Synchronization within the object is implicit in the serial processing of incoming messages.

- *Should object operations be asynchronous or synchronous?*

  Act 1 and Multilisp have only asynchronous object operations, while POOL-T is entirely synchronous. ConcurrentSmalltalk supports both. The advantage of having asynchronous operations is that it allows inter-object parallelism to be naturally expressed; interacting objects proceed in parallel without programmer intervention. Synchronous operations are easier to reason about, but require that the programmer create a new thread of execution to gain concurrency.

  Both synchronous and asynchronous operations can be realized using PRESTO. Synchronous operations essentially come "for free" from from the underlying sequential programming model. An asynchronous operation having no return value (or none of interest) is possible by creating a new thread and using it to request the operation. Later synchronizing on an asynchronous operation requires extra handling.

  In Multilisp and Act 1, an asynchronous operation is described by a *future*, which is an object whose value is either "in progress" or "ready." If a future's value is referenced in an expression, the thread executing the expression is blocked until this value becomes ready. Using PRESTO as a base, a future is represented by an instance of a class derived from the class *Future*. When a future is declared, a new thread is created, and the computation represented by the future executes asynchronously to the thread running in the object holding the future's reference. This reference may be passed around freely. The thread computing the future terminates with a return value that becomes the future's own value. Upon termination, the future has been reached, and any threads waiting on the future's value are resumed.

- *Should any parallel program try to use all of this?*

  PRESTO provides the basic tools with which to construct different types of parallel programming models. While misuse (or abuse) of these tools can be discouraged, it cannot be prevented. Application builders will fail to realize *any* model if they try to realize them all.

5.3. PSIM – An Environment For Parallel Simulation

PSIM provides a message-oriented programming environment for writing parallel simulations of the type described by Chandy and Misra [4] and Bryant [3]. A simulation is structured as a set of processes that communicate by sending timestamped messages to one another. Each process is guaranteed to receive its messages in monotonically increasing timestamp order. The PSIM environment's primary function is to guarantee this ordering, *without* requiring that all processes run lockstep in simulation time. In this way, processes that do not have inter-message dependencies can proceed in parallel.

The PSIM Abstractions

PSIM presents itself as a standalone support system for parallel simulations. Programmers are unaware that the foundation on which PSIM implements its abstractions is provided by PRESTO. Instead of threads, spinlocks, and synchronization objects, PSIM programs manipulate higher-level abstractions such as *Logical Processes* (LP), *links* and *messages*. LPs communicate with one another by sending timestamped messages across links, which are one-way communication channels guaranteeing that all messages are received in increasing timestamp order.

In the same way that synchronization semantics are absent from PRESTO synchronization objects, PSIM's LPs don't simulate anything. Rather, they are a basic class from which more useful simulation processes can be derived. An LP is an object with its own thread of control and its own private simulation clock. It can open links, and send and receive messages on them. Every message includes the clock value of the source LP at the time the message is sent. A clock advances in response to the messages that an LP receives, implying that clocks in different LPs can progress at different rates. The main job of PSIM is to ensure that this variability is transparent to the LPs.

A message cannot be received on a link by an LP unless no other logically earlier message will arrive on any of the LP's other links. This constraint causes LPs to block *even though* there may be pending messages on some of its links.

Handling Deadlock

A PSIM simulation can deadlock if a cycle of empty links exists among a subset of LPs. If the subset is proper, then the system is partially deadlocked and some LPs can continue to make progress. If it is complete, then all LPs are blocked and nothing can proceed without external intervention. To recover from deadlock, the blocked LP with the earliest pending input message is allowed to receive that message.

Part of the research associated with PSIM is the investigation of different methods of handling deadlock (partial and full) in parallel distributed simulations. The structure of PRESTO and, in particular, of PRESTO's scheduler, simplifies the mechanics of this research. When the scheduler concludes that there are no threads to be run (all LPs are deadlocked), it invokes the operation *halt* on itself. PSIM defines a simulation scheduler that is exactly the same as the PRESTO scheduler, *except* for its halting criteria. The simulation scheduler finds the set of all LPs that are blocked but have unread messages on their links. If any exist, the scheduler starts the one with the earliest unread message.

Dealing with deadlock only when the system halts does not solve the problem of partial deadlock. As long as a simulation fully utilizes all of the *physical* processors in the system, a deadlocked subset of LPs does not affect a parallel simulation's performance. Only when there are fewer non-deadlocked LPs than physical processors is the simulation "in trouble."

The typical way of dealing with partial deadlock is to use low-priority threads that find, and break, the deadlock when it occurs. A processor that can't find an LP to execute runs one of these threads instead. Although functionally correct, this solution incurs a scheduling and context-switch overhead. This overhead can be avoided, however, by changing the behavior of the low-level system components.

PRESTO encourages a solution in which the processors themselves solve the deadlock. When a processor object requests a ready thread from the scheduler, the scheduler invokes the operation *get* on the pool of ready threads. With this, the locus of control moves from the processor object to the scheduler object to the ready pool. PSIM supplants the scheduler's ready pool with one of its own. A *get* operation on a PSIM ready pool that would otherwise return nothing, instead scans the set of blocked LPs to determine which, if any, should be restarted. The algorithms for making this determination are wholly enclosed within PSIM's own ready pool. The processor and scheduler objects are essentially duped into breaking partial deadlock without their knowledge.

PSIM and PRESTO together prove that efficiency and abstraction need not be incompatible. The difference between a PRESTO thread and a PSIM LP is one of semantics, not performance. Similarly, links are built using the basic thread primitives (*sleep/wakeup*), and are not constrained by an abstraction that obscures their goals. The combination of object-oriented programming with PRESTO's open system design allows a very high-level concept, namely deadlock detection, to execute efficiently at a very low level. Currently, PSIM is serving as the implementation base for another parallel programming environment, namely Poker [15].

6. The PRESTO Implementation

PRESTO is implemented in the object-oriented programming language C++ [16]. The system runs on a Sequent Balance 21000 shared-memory multiprocessor on top of the DYNIX [17] operating system, and on single-processor DEC VAX machines running the ULTRIX operating system. The system should soon be operational on the DEC SRC Firefly, an experimental prototype multiprocessor workstation.

Sequent's DYNIX is a UNIX-lookalike. The only way to achieve true multiprocessor parallelism is to create multiple DYNIX processes, a fairly expensive task requiring about 55 msec[2]. In contrast, a PRESTO thread on the Sequent can be created and started in as little as 700 μsecs. A large percentage of these times is spent acquiring the atomic hardware locks needed to guarantee mutual exclusion in the various system components (about 30 μsecs. per lock). We expect the Sequent implementation to speed up considerably when the new Symmetry hardware makes it possible to acquire free locks in only a few μsecs. It is encouraging that our design, which invites extension and modification, has performance comparable to that of several other multiprocessor *threads packages* known to us.

In this paper we have concentrated on the way in which PRESTO's object orientation provides a framework within which one can easily build efficient support for a wide variety of parallel programming models. PRESTO's implementation and performance are described more fully in a companion paper [2].

7. Conclusions

PRESTO is not a toy. It is the current system of choice for parallel programming at the University of Washington.

PRESTO began merely as an effort to address the high cost of the parallel programming constructs provided in the DYNIX environment, where we found that our use of threads had to be governed by their overhead rather than by the natural decomposition of our problems. PRESTO succeeded in this goal.

After a significant period of use, though, we have come to the conclusion that PRESTO's ability to be customized to provide efficient support for any of a wide variety of parallel programming models is of much greater importance. Correct and efficient parallel programs are notoriously hard to engineer. There is no one parallel programming model that is right for all applications. The ability to construct an appropriate model using PRESTO makes correct and efficient programs less difficult to achieve.

Acknowledgements

We'd like to thank Kenneth Almquist, Tom Anderson, Jeff Chase, Bjorn Freeman-Benson, Ellen Ratajak, Alan Shaw, and Ken Whaley for their input on the system's design and implementation as well as their many helpful comments on earlier drafts of this paper.

---

[2] Some timings for other Sequent operations: execute one iteration of a for-loop: 4 μsecs.; make a procedure call with no arguments: 15 μsecs.

References

1. P. America, "POOL-T: A Parallel Object-Oriented Language," in *Object-Oriented Concurrent Programming*, ed. M. Tokoro, A. Yonezawa, MIT Press, Cambridge, Mass, 1987.
2. B.N. Bershad, E.D. Lazowska, and H.M. Levy, "PRESTO: A System For Object-Oriented Parallel Programming," Technical Report TR 87-09-01, Department of Computer Science, University of Washington, (submitted for publication), September 1987.
3. R.E. Bryant, "Simulation of Packet Communucations Architecture Computer Systems," Technical Report MIT, LCS, TR-188, Massachusetts Institute of Technology, Laboratory for Computer Science, 1977.
4. K.M. Chandy and J. Misra, "Asynchronous Distributed Simulation Via A Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, no. 11, pp. 198-206, ACM, November 1981.
5. E.W. Dijkstra, "The Structure of the 'THE'-Multiprogramming System," *Communications of the ACM*, vol. 11, no. 5, pp. 341-346, ACM, 1968.
6. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
7. R. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transaction on Programming Languages and Systems*, October 1985.
8. C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 11, pp. 666-677, ACM, August 1978.
9. C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549-557, ACM, October 1974.
10. R. Holt, "A Short Introduction To Concurrent Euclid," *SIGPLAN Notices*, vol. 17, pp. 60-79, May 1982.
11. H. Lieberman, "Concurrent Object-Oriented Programming in Act 1," in *Object-Oriented Concurrent Programming*, ed. M. Tokoro, A. Yonezawa, MIT Press, Cambridge, Mass, 1987.
12. *Modula2+ Reference Manual*, Digital Equipment Corporation, April 1986.
13. D.A. Mundie and D.A. Fisher, "Parallel Processing in Ada," *IEEE Computer*, pp. 20-25, August 1985.
14. B.W. Lampson, D.D. Redell, "Experiences with Processes and Monitors in Mesa," *Communications of the ACM*, vol. 23, no. 2, pp. 104-117, ACM, February 1980.
15. L. Snyder, "Parallel Programming and the Poker Programming Environment," *IEEE Computer*, vol. 17, no. 7, July 1984.
16. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, March 1986.
17. S.S. Thakkar, P. Gifford, and G. Fielland, "Balance: A Shared Memory Multiprocessor," *Proceedings, 2nd International Conference on Supercomputing*, Santa Clara, May 1987.
18. W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications of the ACM*, vol. 17, no. 6, pp. 337-345, ACM, June 1974.
19. Y. Yokote and M. Tokoro, "Concurrent Programming in ConcurrentSmalltalk," in *Object-Oriented Concurrent Programming*, ed. M. Tokoro, A. Yonezawa, MIT Press, Cambridge, Mass, 1987.
20. A. Yonezawa and M. Tokoro, "Object-Oriented Concurrent Programming: An Introduction," in *Object-Oriented Concurrent Programming*, ed. M. Tokoro, A. Yonezawa, MIT Press, Cambridge, Mass, 1987.

The Amber System: Parallel Programming on a Network of Multiprocessors

Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska,
Henry M. Levy, and Richard J. Littlefield

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

This paper describes a programming system called *Amber* that permits a single application program to use a homogeneous network of computers in a uniform way, making the network appear to the application as an integrated multiprocessor. Amber is specifically designed for high performance in the case where each node in the network is a shared-memory multiprocessor.

Amber shows that support for loosely-coupled multiprocessing can be efficiently realized using an object-based programming model. Amber programs execute in a uniform network-wide object space, with memory coherence maintained at the object level. Careful data placement and consistency control are essential for reducing communication overhead in a loosely-coupled system. Amber programmers use object migration primitives to control the location of data and processing.

1 Introduction

Small-scale shared-memory multiprocessors are becoming widely available in implementations ranging from single-user workstations to mini-supercomputers. The proliferation of multiprocessors means that local area networks of these systems are likely to become common. This presents the opportunity to program a group of these machines to work together on a single application. For many applications, networks of small-scale multiprocessors will have greater performance potential than the fastest mainframes at significantly lower cost, and with greater flexibility.

Amber was designed to take advantage of this trend by supporting the development of parallel applications that use multiple machines in a network of shared-memory multiprocessors. Amber provides a set of programming facilities and abstractions that isolate the programmer from the low-level details of programming in this environment. The abstractions are intended to simplify communication, distribution, and parallelism, while supporting a dynamic program structure that can express and benefit from locality.

Amber is based on a model of computation in which a collection of mobile objects distributed among nodes in a network interact through location-independent invocation. Amber objects are passive fine-grained entities consisting of private data and a set of public operations that can be locally or remotely invoked. The active entities in the system are *thread* objects, which possess processor state and a runtime stack and can execute on a CPU. A typical application might contain many threads concurrently executing object operations on different processors in a node and on different nodes in the network. The threads in an Amber program execute in a flat network-wide shared object space. Object references can be transmitted across node boundaries and dereferenced on any node with consistent semantics, allowing programs to operate on distributed data structures in a uniform way.

Amber programs are written in an object-based subset of the C++ programming language [Stroustrup 86], supplemented with primitives for thread management and object mobility. The system is composed of a preprocessor to C++ and a runtime kernel which is linked with the user's program. Amber is implemented on the Topaz operating system for the DEC Firefly [Thacker et al. 88], a multiprocessor workstation based on VAX microprocessors. Applications have been executed on a group of eight Fireflies connected by a 10-megabit/second Ethernet.

This material is based on work supported by the National Science Foundation (Grants CCR-8611390, CCR-8619663, CCR-8700106, CCR-8907666,and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center, the External Research Program, and the Graduate Engineering Education Program).

1.1 Research Goals and Issues

Amber explores issues in parallel programming at both the system level and the application level. At the system level, Amber explores the viability of using a loosely-coupled network of small-scale multiprocessors as a large-scale machine. This raises issues in scheduling, virtual memory management, distribution, and coherency. At the application level, Amber explores how to structure an application to benefit from the target architecture. Here we wish to understand the programming primitives needed to express both locality and parallelism. Overall, Amber assesses the appropriateness of the object-oriented programming model for solving problems at both levels.

A major goal of the Amber project is to provide language-level support for concurrency and distribution using an existing programming language and operating system. This reduces the development cost of the system and makes it more attractive to programmers. We were able to achieve this goal using the facilities of the Topaz operating system and the C++ programming language. Topaz provides useful network services, support for threads, and remote procedure call [Birrell & Nelson 84]. The extensible class hierarchy of C++ enabled us to implement Amber without modifying the language or its compiler.

Amber's programming model assumes that the user will be aware of the network organization and will wish to take advantage of it to achieve maximum application speedup. This conflicts with the goal of network transparency. The tension between uniformity and performance is more pronounced in Amber than in other distributed systems because high performance for parallel applications is the primary purpose of the system. Good performance on a loosely-coupled multiprocessor demands careful attention to data placement in order to minimize remote references, which are three to four orders of magnitude more expensive than local ones. Our view is that data placement should be under the control of the program rather than the runtime system since the needs of different applications will vary widely. This compromises the uniformity of the programming model because it requires the programmer to deal explicitly with location. Amber attempts to strike a balance between uniformity and performance. The programming model is designed to isolate areas where the programmer must be concerned with location, while providing means to tune program organization for efficient execution.

1.2 Related Systems

Object-based models have frequently been used for distributed systems; examples include Hydra [Wulf 74], Clouds [Allchin & McKendry 83], Argus [Liskov 88], Eden [Almes et al. 85], and Cronus [Schantz et al. 86]. Systems such as Eden and Argus provide support for distributed objects at the programming language level using a special-purpose language. Experiments with this approach at the University of Washington led to the development of Emerald [Black et al. 87], a distributed programming language with support for fine-grained object mobility. Amber's distribution model and mobility primitives are derived from Emerald.

In contrast to these other systems, the goal of Amber is to execute a single application that performs a parallel computation, computes a result, and terminates. Amber does not support persistent objects, primitives for reliable distributed computing, or communication and cooperation between unrelated programs. In this respect Amber is related to object-based systems for concurrent programming on tightly-coupled machines. Amber is in fact a direct descendent of one such system, Presto [Bershad et al. 88a], a C++-based runtime package for building medium-grained parallel applications on shared-memory multiprocessors. Amber's thread model and synchronization model follow those of Presto.

Amber was developed to allow a network of machines to be treated as a loosely-coupled multiprocessor using a programming model based on distributed objects. Recent systems with similar goals include Sloop [Lucco 87] and Orca [Bal & Tanenbaum 88]. Amber differs from these systems in that its programming model and internal structure are designed to take advantage of shared-memory multiprocessors. Amber supports logical concurrency and true parallelism within each node as well as across nodes in the network. Concurrency within a mutable object is realized by placing the object on a single node and clustering all threads manipulating the object onto that node. This allows the consistency of the object's internal state to be efficiently managed by hardware-based synchronization primitives and memory coherence protocols. This organization of Amber programs into closely-cooperating clusters is similar to the task force structure in Medusa [Ousterhout et al. 80] and StarOS [Jones et al. 79], but in Amber this clustering is determined at run time and can change dynamically as the computation progresses.

Other researchers have investigated the use of a network as a loosely-coupled multiprocessor within the context of more traditional programming models. The Ivy system [Li 86] pioneered the use of network-wide shared virtual memory for this purpose. This approach allows distributed applications to be written using conventional programming techniques. Ivy maintains memory coherence by using virtual memory hardware to implement page ownership schemes analogous to hardware cache consistency protocols. One goal of Amber is to explore the relative merits of object-based versus shared-memory models for maintaining memory coherence in a parallel and distributed environment. This issue is discussed in Section 4.

2 The Programming Model

Amber provides the programmer with a set of predefined object classes for managing threads, synchronization, and distribution. Amber's abstractions are supplied by means of existing C++ language mechanisms such as subclasses and dynamic object creation. The programming model demonstrates that support for concurrency and distribution can be integrated into a class-hierarchical language to produce a uniform system with features that compose well.

The use of an object-oriented language provides other benefits. Object classes can hide not only the representation of objects but also the internal details of their execution, synchronization, and location. Other researchers have discovered similar benefits for implementing features such as persistence and recovery properties for objects [Herlihy & Wing 87]. Also, dynamically typed subclasses are a convenient vehicle for tailoring system behavior to meet the needs of a specific application. These ideas are made more concrete in the next few subsections, which present the details of Amber's programming model.

2.1 Threads

Amber's mechanisms for expressing concurrency are derived from the thread facilities provided by Presto. Presto was designed to make the use of threads inexpensive, allowing the programmer to efficiently manage more control streams than there are processors. In Amber, threads have the advantage of allowing the programmer to maximize throughput by overlapping computation with remote communication.

Like other objects, threads can be created dynamically using the C++ *new* operator. The basic operations on threads are *Start* and *Join*. The *Start* primitive starts a thread executing an operation on a specified object. *Join* blocks the caller until the specified thread terminates, returning the result from the operation specified in the *Start* call.

Threads provide explicit support for concurrency, in contrast to the implicit support provided by asynchronous object invocation mechanisms in languages such as Sloop. Amber invocations are synchronous, but threads can be used by either the invoking object or the invoked object to transparently provide asynchrony. An invoked object can exploit parallelism transparently to its invoker by creating and starting additional threads in response to an invocation. Alternatively, a thread can execute an asynchronous invocation by creating another thread to perform the invocation.

Amber's scheduler supports timeslicing and can be customized to use priority-based or adaptive policies tuned to the specific application. An application can install a custom scheduling discipline at runtime by replacing the system scheduler object with a similar object that supports the same interface but behaves differently [Bershad et al. 88b].

2.2 Synchronization Objects

Amber provides a flexible set of classes for controlling access to data shared by multiple threads. The system supports relinquishing and non-relinquishing locks, barrier synchronization, monitors and condition variables. Programmers can extend the class hierarchy to define custom mechanisms for concurrency control using these primitive synchronization objects. The intent is that programmers will select an appropriate concurrency control scheme for each user object and encapsulate the details of the synchronization within the class.

Amber's approach to synchronization differs from similar systems designed for networks of uniprocessors. Most of these systems support monitored objects and indivisible operations but no explicit lock primitives. We believe that fine-grained synchronization using lock primitives is desirable when the nodes in the network are multiprocessors. Fine-grained locking reduces contention and allows hardware-based spinlocks to be used to reduce latency when appropriate. Lock objects have additional advantages in a distributed environment because they are mobile and can be remotely invoked to enforce concurrency constraints involving multiple objects on different nodes.

2.3 Controlling Object Location

In Amber, threads invoking operations on an object move to the node where the object resides, so the division of computational load between the machines is determined by the locations of the program's data objects. Object location also has a significant effect on the network overhead incurred by the program. In general, interacting objects should be co-located in order to avoid the cost of a remote procedure call on each invocation. This must be balanced with the need to place objects so as to evenly distribute the computational load between machines.

Amber programmers take advantage of locality by using migration primitives to control object placement as the program executes. Objects can be moved even if they have active invocations: threads executing operations on a moving object are identified and moved with the object. Dynamic mobility is useful because some applications will need to reorganize object locations following different computational phases of a program, although static object placement is sufficient for many applications.

Amber's mobility primitives are modeled after mobility in the Emerald system [Jul et al. 88]. An Amber object can be moved with *MoveTo* and its location can be determined with *Locate*. Like Emerald, Amber provides other mobility primitives that are useful for improving program performance. The programmer can *Attach* an object to another object or *Unattach* an attached object. The attachment primitives allow a programmer to dynamically create structures of objects that move together and are always guaranteed to be co-located. Amber also supports replication of readonly objects to reduce unnecessary communication overhead. Objects may be marked as *immutable* at runtime, indicating that they will never again be modified. Invoking *MoveTo* on an immutable object causes the object to be copied rather than moved. The attachment and immutability mechanisms in Amber are more dynamic than in Emerald, where they are specified at compile time.

Amber leaves all aspects of object location under the direct control of the program. Data objects never move unless the program explicitly moves them. Objects that are not explicitly designated as immutable are never replicated, eliminating the complexity associated with keeping multiple copies of a writable object consistent. This approach contrasts with other recent experiments with language-level support for distributed objects. The Orca language performs automatic object placement and replication of mutable objects, but provides no primitives for explicit object migration. Sloop includes advisory migration primitives, but the system may override the programmer's decisions under certain conditions. Amber attempts to provide a model of sharing and location that is uniform, predictable, and simple to implement. Our assumption is that the best policy for managing location is application-specific and is best left to the program or higher-level object placement software.

3 Implementation Issues

Amber programs execute as a set of cooperating Topaz tasks distributed across the network, with one task on each participating node. The tasks are created at program startup using Topaz facilities for creating remote processes. Each task is an execution of the same program image read from a distributed file system. Topaz supports multiple threads of control in a single task and fast remote procedure call between tasks [Schroeder & Burrows 89], facilities that are used to implement Amber thread scheduling, object migration, and internode object invocation.

The key implementation problem for Amber is the abstraction of a single network-wide object space with object mobility and transparent invocation of remote objects. The following subordinate issues must be addressed in order to implement this model:

- naming, creating, and destroying objects
- moving objects
- trapping nonlocal invocations
- finding remote objects
- migrating threads for remote invocations

Our goal was to implement Amber's shared object abstraction within the confines of C++, using techniques that perform little or no remote communication not directly requested by the program. We found that much of the implementation was straightforward if we used direct virtual addresses as the basis for object naming in the network, arranging each task's address space so that virtual addresses have the same meaning on all nodes. The next few subsections describe our implementation for Amber, with an emphasis on how this global virtual address space is managed and how it simplifies the implementation of the shared object space abstraction. Section 3.5 presents additional problems caused by intranode parallelism, and Section 3.6 discusses our use of the C++ language.

3.1 The Global Address Space

Object references and other pointers are frequently transmitted across the network in Amber. This happens when arguments to a remote invocation are passed by reference, or an object containing embedded pointers moves from one node to another. Also, any thread executing an operation on a moving object will move with the object and resume execution on the destination node, where it will continue to use addresses stored in its stack and registers. The transmitted addresses may be object references, program code addresses, pointers into static data such as string constants, or back links in the stack. It follows that all code and data items are visible to all nodes and may be referenced by any thread regardless of which node it is running on. The references must be resolvable on all nodes with uniform semantics.

One solution to this problem is to translate addresses whenever they cross node boundaries. This is the solution used in Emerald [Jul et al. 88]. Such a scheme permits each node to do independent memory management, which is useful for Emerald because it assumes a universe of long-lived objects created by multiple users. The problem with this approach is that it requires extensive compiler support to aid in the address translation. This is incompatible with our goal of using an existing widely-used language and compiler for Amber.

Amber avoids the need for address translation by ensuring that addresses retain their meaning when transmitted across node boundaries. The global virtual memory is implemented by arranging the virtual address space of each participating Topaz task identically. Program code and statically initialized program data are automatically replicated at the same addresses on all nodes because the tasks are activations of the same program image executed on homogeneous machines. All dynamic objects (including thread objects and their stacks) are assigned a distinct segment of the global address space when they are created, and each object occupies this same virtual address range on any node that it visits during its lifetime. The segment of virtual memory occupied by an object on one node is reserved for that object on all other nodes.

Amber's memory organization requires that nodes use disjoint regions of the address space for heap allocations of dynamic objects. The system must guarantee that two nodes do not attempt to allocate the same heap block, but it must do this without the expense of distributed agreement for each object allocation. Each node is assigned a private region of the virtual address space at startup time for its local heap allocations. Statically partitioning the entire address space in this way is limiting because objects are not allocated uniformly across nodes. For this reason, a large part of the address space is left unallocated at startup and is handed out later by an address space server as nodes exhaust their initial pool. The cost of extending the address space is not excessive because the regions are large enough (currently 1M bytes) that extensions are needed relatively rarely for applications that are moderate in their use of memory.

3.2 Handling Remote References

Each Amber object is referenced by a virtual address that is valid on any node, but the system must determine whether or not an object is local when it is invoked. To provide this information, each object has an *object descriptor* on every node that indicates whether or not the described object is locally resident. The descriptor may contain other information about the object, such as where to look for it if it is remote. Checks inserted by the preprocessor examine an object's local descriptor on each invocation. If the descriptor indicates that the object is remote, the invocation traps to the Amber kernel and is handled by a remote procedure call that moves the faulting thread to the node where the invoked object resides.

An Amber object is implemented as a record, the first part of which is its descriptor, and the remainder of which is its representation (the data local to the object). The virtual address of an object is therefore the address of its descriptor. When a new object is created it is allocated from the heap on a particular node. The descriptor for the object is initialized on that node to indicate that the object is resident so that it can be invoked. If a mutable object is moved, its descriptor is changed to indicate that it is not resident, and a forwarding address is inserted in the descriptor.

Objects and their descriptors are managed so that an uninitialized descriptor is detected and interpreted to mean that the object is remote. This eliminates the expense of initializing remote descriptors for a newly created object. An uninitialized descriptor is detected because unwritten pages of virtual memory are zero-filled by the Topaz operating system, and object descriptors are defined so that the resident flag is a one-valued bit. References to objects occupying heap blocks that were previously deallocated and reused are also handled correctly. This requires that the heap allocation algorithm be constrained so that heap blocks are never divided once they have been returned to the free pool.

3.3 Locating Mobile Objects

When the kernel handles a trap on an invocation of a remote object, it retrieves a forwarding address [Fowler 85] from the object's local descriptor. The forwarding address is left in an object's local descriptor when the object moves away from a node. The forwarding address may be out of date if the object moves frequently. In this case the object's location can be determined by following a chain of forwarding addresses, since the object leaves a new forwarding address on each node that it visits. It is costly to locate an object by following a forwarding chain, but this happens rarely because the object's last known location is cached on all nodes along the chain so that the object can be located quickly on subsequent references.

The situation is more complicated in the case of a trap on an object with an uninitialized descriptor, indicated by the presence of a null forwarding pointer. Each task has complete knowledge of the assignment of heap regions to nodes because a reference to the node that owns each heap region is obtained from the address space server when the region is first mapped by a task. This allows the system to use a heap object's virtual address to identify the object's *home node*, the node on which it was created. When a reference to an object with an uninitialized descriptor is detected, the kernel forwards the request to the object's home node. The home node can determine where the object resides by following the chain of forwarding addresses.

3.4 Object and Thread Migration

The global virtual address space simplifies object migration because it avoids the need to translate addresses stored in the moving object. Furthermore, there is no need to allocate space on the target node for the object since the address range that it will occupy is predetermined. Moving an object involves copying its contents from the source node to the destination node and updating its descriptors on both nodes. The implementation is complicated by the need to identify and move threads that are actively executing operations on the object. These *bound threads* must migrate with the object in order to preserve the consistency of the object's contents. This problem is discussed in Section 3.5.

Amber remote invocations are performed by simply moving the invoking thread to the remote node. In principle this is no more complicated than any other object move. The thread's control information and pieces of its stack are copied to the same address ranges on the remote node, the object descriptors are updated, and the thread is added to the scheduling queue on the remote node. Addresses in its processor registers and stack will continue to be valid on the destination node. In practice, thread migrations are handled slightly differently from migrations of other objects in order to optimize remote invocations made by the thread at the expense of invocations made on the thread object itself (e.g., a *Join* operation).

3.5 Object Mobility on Multiprocessors

Dynamic mobility is difficult to implement on multiprocessors because user threads may be attempting to manipulate a moving object concurrently with the mobility code running on another processor. This leads to a number of implementation concerns that would not arise on a uniprocessor. In uniprocessor object migration all threads on the node are implicitly suspended while mobility code in the kernel is in control of the single processor, making it a simple matter to determine which threads are bound to the moving object by examining their stacks. Furthermore, the system can preserve the atomicity of the invocation sequence by preempting threads only at safe points in their execution, avoiding a context switch between the residency check and the completion of the stack modifications indicating that an invocation is active.

On multiprocessor hardware the atomicity of descriptor checks can no longer be guaranteed. In a naive implementation a thread could check the descriptor and find that the object is local, but not actually complete the local invocation until after a move operation on the object has been initiated by another processor. This race condition will always exist if the descriptor is checked before the stack modifications associated with the invocation are made. An analogous race condition can occur on returns: a thread could check that an object is still resident before its return, only to have the object move after the check but before the actual control transfer. A related problem is that the set of active threads bound to a moving object is constantly changing while the mobility code is running.

One approach to solving these problems is to lock the invocation sequence and maintain a data structure that records which threads are currently executing within each object. This solution makes invocations expensive because of the need to synchronize and update the data structure. Another approach is to freeze all activity on the node during a move operation and examine the stacks of all local threads to determine which threads are bound to the moving object. This solution optimizes invocations but makes move operations complex and expensive. There are many gradations between these extremes.

Amber makes invocation-time residency checks at the start of each operation, after the invocation stack frame is pushed but before any user code is executed. Return-time checks are made immediately after the invocation frame that the thread is returning from has been popped. This guarantees that the executing thread can be identified as bound to the object before it actually checks the descriptor and enters the object. Threads already bound to a moving object are handled by an additional residency check that is made on each context switch into a preempted thread. Move operations in Amber preempt and reschedule all threads running on the source node, forcing them to make a residency check before they continue. The preemptions occur after the descriptor of the moving object has been marked as non-resident but before the object's contents have been copied to the remote node.

Local invocations are efficient with this scheme because they require no synchronization over the object descriptor, only a residency check consisting of a single VAX branch-on-bit-set instruction. Also, there is no need to halt all activity on a node during a move operation; at worst it will be necessary to briefly interrupt each processor. One problem is that some concurrency may be lost if the destination node is idle but the source node is busy, since suspended threads which are bound to the object will not move to the destination node until they are rescheduled on the source node. An added disadvantage is that the need to preempt all running threads causes the cost of mobility to increase as processors are added to a node. The assumptions behind these tradeoffs are (1) object moves are much less frequent than object invocations, and (2) improvements in processor speeds will make thread preemptions cheap relative to the network latency associated with a move operation.

3.6 Experience With C++

Our choice of C++ was partly motivated by its availability and its popularity with programmers. Another advantage of C++ is that it is efficiently implemented with a minimum of runtime support. Most other benefits of using C++ could have been obtained from any object-oriented programming language with an extensible class hierarchy and dynamic typing.

In our Amber prototype, object descriptors are allocated and managed by deriving all user classes from a single base class called *Object* whose private data items include the descriptor. The constructor and destructor functions for the *Object* class maintain the descriptor and ensure that object creation and deletion meet the requirements discussed in Section 3.3. Threads and synchronization objects are provided by introducing new subclasses of *Object*. The mobility primitives are operations on instances of class *Object*. Amber's distributed heap allocation is implemented by redefining the runtime library routines for the C++ operators *new* and *delete*.

One problem with C++ is that Amber's distribution model depends on the regularity of an object-oriented programming language. Amber assumes that a thread will never directly manipulate the internals of a remote object, since references to remote objects are recognized and trapped only on invocations. Furthermore, all data items that may be referenced remotely must be encapsulated in an object. The C++ language includes many performance features that circumvent constraints normally associated with an object-oriented programming model. Examples of such features are friends, public member elements, inline functions, unprotected structures, and the ability to include arbitrary C code in the program. These features can result in incorrect program behavior if they are used improperly in a distributed environment. Nevertheless, they present opportunities to optimize interactions between objects that are known to reside on the same node.

There are a number of situations in which co-residency guarantees make it possible to use these features safely. Co-residency can be explicitly requested using Amber's attachment primitives. Also, C++ *member objects* (objects that are directly contained within some other object) always move with their containing object and are therefore co-resident with it. Co-residency guarantees can also be exploited to optimize invocations of functions in base classes or invocations of objects allocated from a thread's stack. Intelligent use of the performance features of C++ in situations where co-residency is assured can significantly improve program performance. For example, consider a multi-threaded object whose internal state is protected by a non-relinquishing lock. If the lock is a member object of the protected object then it can be safely acquired and released using fast inline function calls.

4 Comparison with Shared Virtual Memory

This section explores the relationships between page-oriented and object-oriented shared memory models. Both approaches offer uniformity relative to an RPC-based programming model, but they differ in other respects. The original motivation for an object-oriented memory in Amber was that objects are a natural unit for involving the programmer in data placement decisions. In this section we shall argue that the object is also a natural and efficient unit for maintaining coherence of the global address space, and that object-level coherence has a number of advantages over page-based coherence.

The memory organization of a loosely-coupled system is closely related to issues of consistency of the data shared by multiple nodes. At the hardware level each node can address only its private physical memory. Coherence of these private memories is difficult to maintain efficiently in a distributed environment. A similar problem is encountered by the designers of programming support for NUMA multiprocessors, where the varying costs of referencing different areas of memory motivate the use of caching, replication and data migration to improve program performance. NUMA programming systems such as PLATINUM [Cox & Fowler 89] make hidden data placement and replication decisions while presenting the programmer with a view of memory that is uniform and coherent at the byte level. This approach can work well for NUMA multiprocessors because the cost of a poor placement decision is typically not very high.

Similar shared memory models have been used to allow a network of machines to be programmed as a loosely-coupled multiprocessor. In Ivy, distributed processes execute in a global virtual address space with consistency of arbitrary bytes guaranteed across references from multiple nodes. Coherence of the shared memory is maintained by memory managers on each node, which use page faults to detect shared accesses and exchange coherency messages with other memory managers [Li & Hudak 86]. Remote references are handled by moving or copying the referenced page to the location of the faulting process. Distribution and load balancing are achieved by explicit process migration.

Amber represents an alternative vision of uniform and consistent memory in which the granularity of data coherence is the object rather than the individual byte. These systems present the programmer with a network-wide object name space, with consistency maintained by trapping invocations of remote objects. This memory model is uniform in the sense that it is unnecessary for the programmer to deal explicitly with the locations of objects when they are invoked, but it is more restrictive than the shared virtual memory approach because it requires adherence to an object-oriented programming discipline.

4.1 Function Shipping

A major difference between Amber and Ivy is that Amber takes a function-shipping rather than a data-shipping approach to coherence. Instead of attempting to maintain the consistency of mutable objects across references from multiple nodes, each object is placed on a single node where access to it is controlled through its operations. Function shipping is especially attractive when the nodes in the network are shared-memory multiprocessors because it clusters the threads referencing a given object onto the same node, where hardware-based synchronization and memory sharing can be used to their fullest performance advantage. The programmer of a data-shipping system such as Ivy can obtain the same advantages through an appropriate use of explicit process migration.

Distributed synchronization is simple and efficient in a function-shipping system. For example, Amber locks are objects which can be remotely invoked to synchronize threads executing on different nodes. References to a shared lock variable can cause a data-shipping system to thrash by repeatedly shuttling the page containing the lock variable between the nodes which are referencing it. Recent versions of Ivy have handled this problem by deviating from the data-shipping model and accessing shared lock variables with remote procedure calls.

For a certain class of programs the behavior of the function-shipping approach is more predictable than that of the data-shipping approach. It is easy to predict the communication overhead incurred by an Amber program that utilizes static object placement or that moves objects at well-defined points. A similar program for a data-shipping system can thrash when a memory page is repeatedly referenced by processes on different nodes. The Amber program can thrash when a thread repeatedly invokes the same remote object, but this effect is less dependent on the orders of events and the timings of concurrent operations (except those involving explicit object moves). In such a program the location of an object can be determined from the program structure and is independent of which threads happen to be referencing the object at the moment.

4.2 Pages vs. Objects

The performance of a coherence policy is dependent upon the degree to which memory references made by the program are localized within the units used by the system to maintain coherence. In a distributed object system the granularity of coherence is the data object, a problem-oriented unit, whereas in shared memory systems it is the page, a unit that is dependent upon the hardware rather than the structure of the program.

The performance of a page-based coherency scheme may suffer if the sizes of data items do not match well with the page size. If a remote data item is larger than a page, an operation that accesses the item in its entirety will generate multiple page faults unless the process is explicitly moved to the location of the data item. In Amber, the thread moves to the location of the data item and the operation executes with a single network transaction. Alternatively, the Amber programmer can choose to migrate the object explicitly, making use of an efficient bulk transfer protocol.

If data items are smaller than a page, a page-based coherency scheme incurs unnecessary communication overhead when logically unrelated data items that happen to reside in the same page are referenced repeatedly by multiple nodes. The programmer for such a system must be aware of page sizes and boundaries to reduce this artificial sharing, just as programmers of current shared-memory multiprocessors need to be aware of cache line sizes in order to achieve the best performance. Page-based systems can reduce these problems by depending on the compiler to structure the data appropriately. This structuring comes for free in an object-based system.

Another argument for object-level coherence is based on a hypothesis that the memory reference patterns of object-oriented programs are more localized than similar programs using more traditional models. The body of an object operation can reference only the thread stack and the contents of the object itself, so an executing operation is likely to make a sequence of memory references local to the current object. In effect, there is knowledge implicit in the way the data area is divided into objects that can be exploited to make the coherence algorithm more efficient.

5 Cost of Amber Operations

The true test of Amber's performance is the behavior of applications built with the system. Section 6 describes a simple application and discusses its performance. It is also useful, though, to measure the cost of the primitives for concurrency and distribution. Table 1 presents some timings for basic Amber operations, as measured on Firefly workstations with four CVAX processors available for running user threads. The latency of these operations is highly sensitive to a number of factors, but the benchmarks that produced these timings attempt to measure the cost of the operations in the most common case. For example, the benchmarks assume that all moving objects and threads will fit in a network packet, and that the destinations are found by following a forwarding chain for one hop. These timings should be regarded as rough indications of the cost of the operations under light load conditions. Operations involving thread scheduling or network communication are more expensive on a heavily loaded system.

Operation	Latency (ms)
object create	0.18
local invoke/return	0.012
remote invoke/return	8.32
object move	12.43
thread start/join	1.33

Table 1: Latency of Amber Operations

We expect that the CPU cost of these operations will have less effect on program performance in the future. As processors get faster the CPU overhead of using any distributed system becomes less significant, and the performance of the system is dominated by network latency, which will remain roughly constant despite the advent of new high-throughput networks. The performance of a distributed system is best evaluated not by the cost of basic network operations, but by the degree to which the system prevents unnecessary network communication.

6 An Amber Application

This section presents the structure and performance of an Amber program that computes the steady-state temperature over the interior of a square plate given the temperatures around the plate's boundary. The behavior of this system is governed by Laplace's equation, which states that the value at each point is the average of the values of its neighbors. The algorithm used is Red/Black Successive Over-Relaxation (SOR), an iterative method that parallelizes well and is commonly used in practice [Ortega & Voigt 85]. This algorithm can be understood by analogy to a checkerboard. Each point of the problem grid corresponds to a square on the checkerboard. During each iteration, all of the black points are updated first, followed by all of the red points. After some number of iterations the computed values converge and the algorithm terminates. Black points have only red neighbors and vice versa, so each

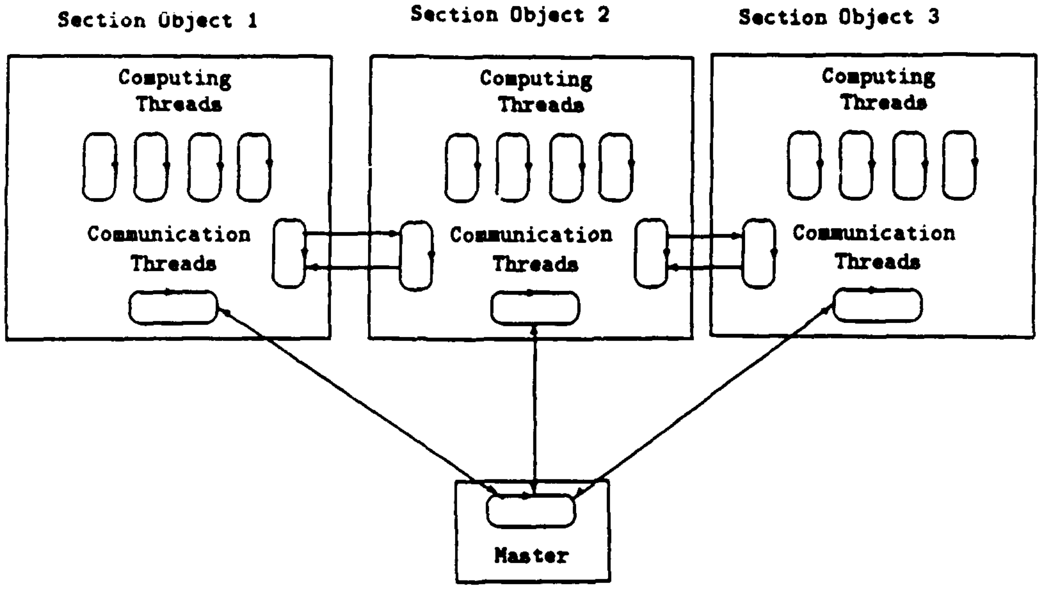


Figure 1: Structure of the Amber Red/Black SOR Implementation

of the update phases is highly parallelizable.

The algorithm is partitioned for loosely-coupled parallel execution by breaking the grid into sections and distributing the sections among the nodes. Some partitionings are clearly inefficient. For example, placing the entire grid in one object would result in unbalanced use of the available processing power. Placing each point in a separate object would involve excessive communication overhead. A more effective approach is to choose the partitioning so that one section object can be assigned to each node. This balances the load and allows the values for an entire edge of a section to be transferred in a single invocation.

The Amber SOR program has several sets of threads associated with each section object. One set of threads computes the values for the section's points in parallel on each iteration. Another set of threads is responsible for exchanging edge data with neighboring sections. The exchange of values for edge points of one color is overlapped with the computation for points of the other color. After each iteration the nodes synchronize at a barrier to determine if convergence has been reached. One additional thread per section is responsible for communicating with a single master thread regarding convergence. Figure 1 displays this structure for a decomposition with three sections.

The SOR algorithm is well-suited to a loosely-coupled multiprocessing model because the problem is regular and static, which makes it easy to choose a partitioning that balances the load evenly. The amount of computing required per section on each iteration depends only on the size of the section and is not affected by the data contained there. Nevertheless, SOR is a nontrivial algorithm which is typical of many iterative methods involving nearest-neighbor interactions. Performance measurements for the program are shown in Figures 2 and 3.

Figure 2 plots measured speedup of the SOR program as the number of nodes and the number of processors increases. For the purposes of this experiment, we selected a specific problem with a grid size of 122 by 842 points. Most of the partitionings were into eight section objects, except for the experiments involving three and six nodes, which were run with partitionings of six section objects. A significant amount of remote communication is required to solve this problem on multiple nodes. Each point in this figure represents the measured speedup for a particular experiment relative to a sequential C++ implementation used as the baseline case. Each point is labeled to indicate the number of Firefly nodes used, and the number of processors per node. For example, the point labeled "4Nx2P" corresponds to an experiment in which the eight sections of the grid were distributed among four Fireflies (two per Firefly) and two processors per Firefly were used (for a total of eight processors). A number of conclusions can be drawn from Figure 2:

- Good speedups are possible in this environment. The SOR program attains a speedup of 25 for the 8Nx4P case – eight Firefly workstations, each contributing four processors to the overall solution.

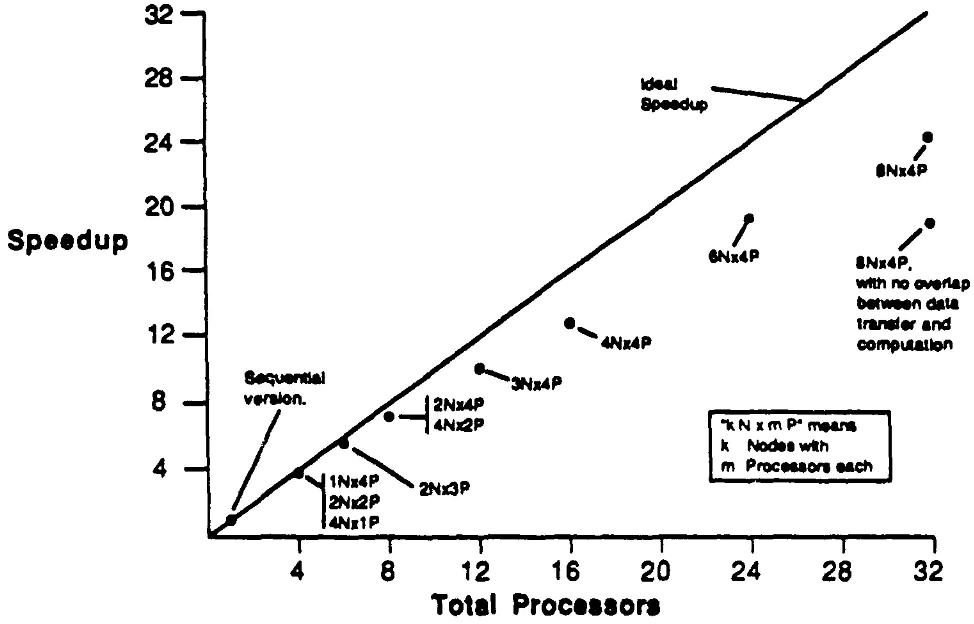


Figure 2: Measured Speedup for Amber Red/Black SOR Implementation

- Significant performance benefit comes from structuring the program so that transfers of edge data are overlapped with computation over the interiors of sections. This is demonstrated by the different performance of the two 8Nx4P cases. This shows the importance of overlapping communication and computation in a loosely-coupled environment.

- The overlapping of communication and computation makes it possible to keep all processors busy doing useful work even while communication is taking place. The performance of this application is not degraded significantly by the cost of remote communication. This is demonstrated by the speedup of the Amber version, which is close to the ideal speedup relative to the sequential version. Also, nearly identical speedups are achieved for all of the experiments involving a total of four processors (1Nx4P, 2Nx2P, 4Nx1P). Similar results were obtained from the experiments with eight processors (2Nx4P, 4Nx2P).

To be fair, the ratio of computation to communication for this program is a function of the grid size. Even if communication is highly efficient, for sufficiently small grids it will dominate computation and limit speedup. For sufficiently large grids computation will dominate and speedup will be good even if communication is relatively inefficient. Figure 3 shows the effect of varying the problem size for the particular configuration of four nodes with four processors each (4Nx4P in Figure 2). The horizontal axis in Figure 3 is the number of points in the grid. The vertical axis gives speedup relative to a sequential version of the program. The point marked "X" corresponds to the 122 by 842 grid used in Figure 2.

We were able to achieve good performance in our Amber SOR program for several reasons. A single network exchange is required to transfer an entire row or column of data between sections, regardless of how data happens to be laid out in the address space. Second, data transfers can be overlapped with computation by running the respective threads in parallel. This reduces the effect of network latency. Third, computation threads within a section can freely divide work among themselves, without danger of causing network activity.

We have not implemented this application under a system with a page-oriented distributed virtual memory, so it is impossible to make exact comparisons with such a system. Certainly a shared memory version under a system such as Ivy would have required less coding effort initially. The performance of the resulting program ultimately depends on how efficiently data can be shared between nodes. The methods for controlling sharing and communication using Amber, with its object-oriented distributed virtual memory, and using a system with a page-oriented distributed virtual memory, are quite different. Using a page-oriented system, the programmer would optimize data reference patterns by laying out data structures and partitioning the work so as to make each node reference different sections of the linear address space. If two nodes write-share the same block of addresses, the virtual memory system will thrash. It may not be obvious from the source code that this can happen. Also, the layout of the data in memory may incur the cost of multiple faults and multiple page transmission latencies to transfer edge data. With Amber the decomposition is addressed explicitly: the programmer has control over what data is transferred and when.

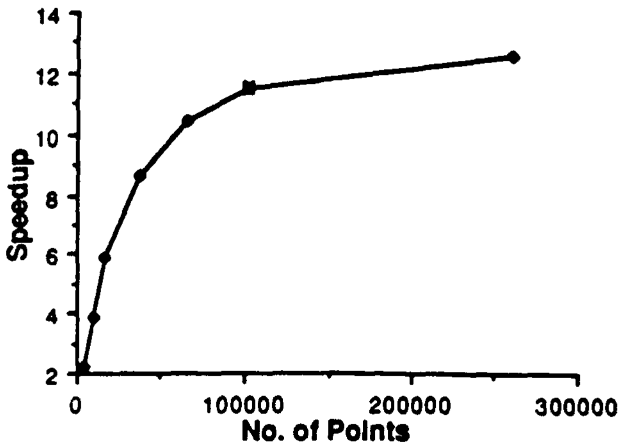


Figure 3: Effect of Varying SOR Problem Size (4Nx4P)

7 Summary

The Amber system permits a loosely-coupled network of multiprocessors to be viewed as an integrated system for executing a parallel application. This underlying hardware architecture is cost-effective for many parallel applications. Processors can be added to a computer system at small marginal cost, but packaging constraints limit the practical size of a single system. Therefore programmers will want to build parallel programs that cross machine boundaries.

With Amber we have shown that the distributed object model is useful for loosely-coupled multiprocessing as well as for distributed programming and distributed operating systems. Amber's object-oriented model strikes a balance between the ease of programming afforded by a page-oriented distributed virtual memory and the performance benefits of explicit management of location. We have achieved a simple and efficient implementation using an existing programming language and an existing operating system. Our application experience thus far indicates that the fundamental goal of Amber - to allow the power of a network of small-scale multiprocessors to be harnessed for a single parallel application - has been achieved.

8 Acknowledgements

Norman Hutchinson and Eric Jul were involved in early discussions of Amber's memory model. Guy Carpenter implemented several pieces of the Amber runtime system. Brian Bershad and Jan Sanislo provided numerous comments and helped with the operating system and hardware of the Firefly. Reid Brown, Tom Anderson, Jeff Bowden, and Ewan Tempero commented on early versions of this paper. Hugh Lauer assisted with final revisions. We would also like to thank the DEC Systems Research Center for providing the Firefly workstations and the Topaz operating system software.

References

[Allchin & McKendry 83] Allchin, J. and McKendry, M. Synchronization and recovery of actions. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 31–44, August 1983.

[Almes et al. 85] Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[Bal & Tanenbaum 88] Bal, H. E. and Tanenbaum, A. S. Distributed programming with shared data. In *Proceedings of the International Conference on Computer Languages*, pages 82–91, October 1988.

[Bershad et al. 88a] Bershad, B. N., Lazowska, E. D., and Levy, H. M. Presto: A system for object-oriented parallel programming. *Software - Practice and Experience*, 18(8), August 1988.

[Bershad et al. 88b] Bershad, B. N., Lazowska, E. D., Levy, H. M., and Wagner, D. An open environment for building parallel programming systems. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming Environments, Applications, and Languages*, July 1988.

[Birrell & Nelson 84] Birrell, A. D. and Nelson, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Black et al. 87] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[Cox & Fowler 89] Cox, A. L. and Fowler, R. J. The implementation of coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[Fowler 85] Fowler, R. J. *Decentralized Object Finding Using Forwarding Addresses*. PhD dissertation, University of Washington, December 1985. Department of Computer Science Technical Report 85-12-1.

[Herlihy & Wing 87] Herlihy, M. P. and Wing, J. M. Avalon: Language support for reliable distributed systems. In *IEEE Fault-Tolerant Computing Symposium Digest*, July 1987.

[Jones et al. 79] Jones, A. K., Chansler, R. J., Durham, I., Schwans, K., and Vegdahl, S. R. StarOS, a multiprocessor operating system for the support of task forces. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 117–127, December 1979.

[Jul et al. 88] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Li & Hudak 86] Li, K. and Hudak, P. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[Li 86] Li, K. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD dissertation, Yale University, September 1986. YALEU/DCS/RR-492.

[Liskov 88] Liskov, B. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Lucco 87] Lucco, S. E. Parallel programming in a virtual object space. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 26–34, October 1987.

[Ortega & Voigt 85] Ortega, J. and Voigt, R. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, pages 149–240, 1985.

[Ousterhout et al. 80] Ousterhout, J. K., Scelza, D. A., and Sindhu, P. S. Medusa: An experiment in distributed operating system structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[Schantz et al. 86] Schantz, R. E., Thomas, R. H., and Bono, G. The architecture of the Cronus distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 250–259, May 1986.

[Schroeder & Burrows 89] Schroeder, M. D. and Burrows, M. Performance of Firefly RPC. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[Stroustrup 86] Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

[Thacker et al. 88] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Wulf 74] Wulf, W. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

Operating System Support for Parallel Computing

Spinning Versus Blocking in Parallel Systems with Uncertainty

John Zahorjan and Edward D. Lazowska

Department of Computer Science
University of Washington

Derek L. Eager

Department of Computer Science
University of Waterloo

February 1988

Abstract

In waiting for an event on a parallel machine, a thread of control may either spin (busy wait) or block (relinquish the processor). The appropriate mechanism depends on the relationship of the expected spin time to the context switch time on that machine.

If the programmer has accurate information about the behavior of an application, the choice between spinning and blocking can be made relatively easily. This might be the case, for instance, when a parallel machine is dedicated to a single, well understood application. However, in the presence of uncertainty, the choice of mechanism is more difficult.

In this paper we examine the choice between spinning and blocking in environments characterized by two kinds of uncertainty: multiprogramming, where the applications programmer does not have control over which threads are running at any point in time, and data-dependent programs, where expected running times can depend heavily on input data. We compare the loss incurred by spinning in these two environments to that in systems running a single, "well-behaved" application. Our goal is to determine how multiprogramming and data-dependent behavior affect expected spin time, and so complicate the job of selecting the right mechanism.

We examine the base, multiprogrammed, and data-dependent environments for two different situations: lock acquisition for mutual exclusion and for barrier synchronization. Using simple analytic models we conclude that for the case of lock acquisition neither multiprogramming nor data-dependent behavior significantly increase the expected spin time, and thus do not complicate the choice of mechanism. However, for barrier synchronization both kinds of uncertainty lead to sharply increased spin times, and thus must be taken into consideration when choosing between spinning and blocking.

Index Terms – Multiprocessors, locking, performance, parallel software, parallel computing.

This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, Digital Equipment Corporation (the External Research Program and the Systems Research Center), and the Natural Sciences and Engineering Research Council of Canada. This work was done while Eager was with the Department of Computational Science, University of Saskatchewan, and while Zahorjan was on sabbatical leave at Laboratoire MASI, University Paris 6.

Authors' addresses: John Zahorjan and Edward D. Lazowska, Department of Computer Science FR-35, University of Washington, Seattle WA 98195; Derek L. Eager, Department of Computer Science University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

1. Introduction

When a thread of control on a parallel machine must wait for some event before proceeding, it may be reasonable for the thread to spin (or busy wait) – that is, to sit in a tight loop continuously checking for the required condition. The time spent spinning is overhead, and since the processor is occupied and cannot be allocated to another thread, the effective processing rate of the system is decreased. An alternative to spinning is blocking – that is, relinquishing the processor. The context switch time required to block also is overhead, so blocking too decreases the effective processing rate of the system.

The appropriate choice between spinning and blocking depends on the relationship of the expected spin time to the context switch time. This choice is not always clear, and a mistake can have major performance implications. For example, one field test release of the DYNIX operating system for the Sequent multiprocessor [Beck et al. 1987] included the substitution of blocking for spinning in a single routine as a "performance enhancement". Under high loads, this change in fact caused a severe performance degradation, something that was first noticed by a Sequent competitor and used as the basis of an advertising campaign [Rodgers 1986].

Although adaptive mechanisms are possible (see, for example, [Ousterhout 1982] and [Lo & Gligor 1987]), the decision of whether to spin or to block is most often made statically at program creation time by the programmer. In highly controlled environments where the programmer has accurate information about the expected spin time, this decision may be straightforward. For example, when the parallel machine is dedicated to a single application that uses locks for mutual exclusion, the programmer may know that a particular lock is held infrequently and for only a very few instructions. Spinning would be the clear choice in this case. Similarly, parallel solutions of large numerical problems often are obtained by partitioning the problem among a number of threads equal to the number of processors. Spinning is the clear choice here, too, since there are no other threads that could run.

The choice between spinning and blocking is not so straightforward when the expected spin time depends on run-time factors. How to make this choice in the presence of such uncertainty is the subject of our paper.

We consider two typical situations in which threads must wait, requiring that either spinning or blocking be employed. The first situation involves waiting because of competition among threads. Here we assume a set of largely independent threads that use a lock to provide mutual exclusion when accessing some resource. When a thread wanting the resource finds the lock in use, it waits until the lock becomes free. The second situation involves waiting because of cooperation among threads. Here we model a set of threads that attempt to synchronize at a barrier. Each thread reaching the synchronization point (that is, the barrier) waits until all other threads have also reached the barrier.

Our "baseline case" is a controlled environment in which the choice between spinning and blocking is straightforward: a single parallel application running on a dedicated multiprocessor with as many processors as there are threads of control.

We consider the effect on this choice of two run-time dependencies. The first is multiprogramming. As parallel architectures become more common, parallel machines and parallel algorithms increasingly will be the choice for general-purpose computing. Clearly, multiprogramming of user jobs is a requirement in this environment. However, because the programmer does not have explicit control over the scheduling decisions made on a multi-user multiprogrammed machine, it is impossible to know which threads of a parallel application will be running at any particular time. Thus, the spin time can be highly variable, depending on whether or not the thread that will eventually generate the event being waited for is currently allocated a processor.

The second source of uncertainty that we consider is data-dependent software. Here we assume that the thread that eventually will generate the awaited event sometimes completes quickly and sometimes runs for a long time. Its behavior depends on its state and the data with which it is presented, and so is not predictable at the time that the application is coded.

The goal of our work is to determine how spin time is affected by these two forms of uncertainty when compared with the baseline case. If the expected spin time is roughly the same in all three situations, the decision between spinning and blocking can be made as if the application were running in a well controlled environment. Since programmers are already dealing with this problem in that environment, this would mean that no new special procedures are required. On the other hand, if the spin time can be significantly lengthened in environments with uncertainty, the programmer's task is greatly more difficult, since it will be necessary to estimate at program creation time parameters that become known only at run time.

Our study is conducted using analytic models validated via simulation. We examine the degradation arising from using spinning under uncertainty relative to that arising using spinning in a controlled environment. We decided against constructing models of performance under blocking with which to compare our spinning models. Information concerning the relative merits of spinning and blocking can be drawn from the spinning models alone, and the results obtained from models of this single type are less sensitive to the precise modelling assumptions made, since any inaccuracies appear consistently. Thus our performance comparisons (if perhaps not the absolute performance values) will be accurate. (See, for example, [Dubois & Briggs 1982] as a contrast in the complexity and flexibility of modelling approaches.)

In Section 2 we describe more precisely the models employed in our comparisons, and we outline briefly the analytic approach. A more detailed discussion of the analysis is found in Appendix A. Section 3 is a discussion of the results for waiting due to lock contention. Section 4 presents the results for waiting due to barrier synchronization. Section 5 contains our conclusions.

2. The System Models

Clearly, a useful performance model must embody enough of the details of the system it represents so that the model's behavior parallels that of the system. At the same time, "unnecessary detail" should be avoided, and the model kept as simple as possible [Lazowska et al. 1984], for at least two reasons. First, simplicity aids in understanding the interaction of the model parameters. A model with many parameters implies an enormous parameter space and consequently a potentially unmanageable set of experiments and results to explore the significance and interaction of those parameters. Second, a simple model is usually more quickly analyzed than a complex one, and so eases the practical burden of running the necessary experiments.

We have constructed two different but similar models, one for lock contention and the other for barrier synchronization. Each model accommodates the three environments (baseline, multiprogramming, and data-dependence) in a natural way. This is important because our goal is to compare spin times between environments, so consistency across those boundaries lends confidence that the comparison is valid.

Performance predictions for our models may be obtained by either simulation or numerical analytic techniques. In fact, we have developed and run software for both approaches. However, all of the results given here are taken from the analytic solutions. Simulation was used only for a sample set of cases with the sole intention of verifying that the analytic software was functioning correctly. The analytic approach is preferable to simulation in this application because the results it provides are exact equilibrium performance measures (rather than stochastic estimates and confidence intervals) and because in general the analytic software is able to obtain results much more quickly than the simulation software.

2.1. Lock Contention

Our lock contention model consists of P identical processors and J threads. We model explicitly the contention for a single lock. Each thread is in one of three states: computing, spinning, and critical section. A thread computes for an average of T time units between attempts to obtain the lock.[1] When a thread requires the lock, if the lock is free the thread immediately acquires it. A thread holding the lock uses it for an average of L time units, then releases it and returns to the computing state. If the lock is not free when requested, the thread spins until the lock is released. If multiple threads are spinning when the lock is released, one of these threads is chosen at random to acquire the lock next.

---

[1] In reality, this means that the thread occupies the processor for an average of T time units between requests. The thread may be performing useful work or spinning on another lock or for some other reason during this time.

For the baseline case of a controlled system, the model is exactly as described above with $J = P$, that is, one thread per processor. This represents the situation in which the machine is dedicated to a single application at a time, and thus presents the application programmer with the least amount of uncertainty regarding the behavior of the software.

A multiprogramming environment is reflected in the model in two ways. First, there are more threads than processors ($J > P$), thus reflecting the fact that, in a multiprogramming environment, an application might at times have more threads than it has been allocated processors. Note that it is most appropriate to increase the number of threads rather than decrease the number of processors when deriving a multiprogramming instance of the model from a baseline instance, because the inherent degree of lock contention is thereby kept constant (as this depends on the total instruction delivery rate as determined by the number of processors, not on the number of threads), and thus any changes in performance can be attributed solely to the effects of multiprogramming.

At any one time, P threads are *scheduled* (allocated processors) and $J-P$ are *unscheduled* (without processors). The second way in which multiprogramming must be reflected in the model is the introduction of a scheduling rule that controls which threads fall into each category. We define a parameter Q that represents the mean scheduling quantum. Each scheduled thread is allowed to use its processor for an average of Q time units before it is unscheduled. (The scheduling of threads on any particular processor is independent of that on all other processors.) When a thread's quantum expires, its processor is assigned at random to a currently unscheduled thread.

This random scheduling is the major simplification of our model, as the replacement policy in a real system is more likely to be FCFS in nature. However, note the mean time that a thread remains unscheduled between successive uses of a processor is identical under random and FCFS replacement, as is the mean amount of computing provided to each thread per time unit. Thus, intuitively we expect the mean performance measures observed under the two scheduling disciplines to be similar.

The primary motivation for assuming random replacement is that it enormously simplifies the model state space. In particular, our model has $2P(J-P)+2J-P+1$ states while an identical model with FCFS scheduling has more than 2^{J-P}. This simplification not only allows results to be obtained more quickly, but also permits the analysis of models with larger numbers of threads and processors than could be examined otherwise.

To model data-dependent behavior we again let J equal P, that is, the model is not multiprogrammed since we wish to isolate the particular effect of data-dependence. However, here we let the lock holding time be highly variable. In particular, with probability $1-p$ a thread acquiring the lock releases it instantly, and with probability p holds it for mean time $\frac{L}{p}$. This results in a mean holding time of L, just as in the baseline case, but with much greater variance.

2.2. Barrier Synchronization

The model of barrier synchronization is quite similar to the lock contention model. In the baseline case there are P processors and $J = P$ threads. Each thread is in one of two states: computing or spinning. A thread computes for an average of T time units before reaching the barrier. If not all other threads have already reached the barrier, it begins to spin. When the last thread reaches the barrier, all threads return to the computing state.

For the multiprogramming environment, we keep constant at P the number of threads involved in the barrier, but add K additional threads. These threads are always in the compute state, but their presence on the processors interferes with the progress of the P "barrier threads". As previously, Q is the mean scheduling quantum and random replacement among the $J+K-P$ unscheduled threads is used as the scheduling discipline.

It is important in this model that we keep the number of barrier threads the same as in the baseline case. The mean time to reach a barrier increases naturally with the number of threads involved in the synchronization. Since we are trying to isolate the effect of multiprogramming, it would not be suitable to simplify the model further by having all $P+K$ threads be involved in the barrier, as it would be difficult to separate the increased spin time due to multiprogramming from that due to the increase in the number of barrier threads. (Note that this is in contrast to the lock contention situation, where the inherent lock contention is determined not by the number of threads but by the instruction delivery rate as determined by the number of processors.)

To reflect data-dependent behavior in the model, K is zero as in the baseline case (i.e., there are only the $J = P$ barrier threads), but there is much greater variance in the compute time required before a thread reaches the barrier. With probability $1-p$, a thread computes for zero time units before reaching the barrier, while with probability p it computes for an average of $\frac{T}{p}$ time units. This maintains the average compute time of T time units, as in the baseline case, but greatly increases the variance.

A more detailed discussion of the analytic formalities of these models can be found in Appendix A.

2.3. Choosing Parameter Value Settings for the Model

A problem that must be confronted immediately in attempting to compare spin times among the three environments (base, multiprogramming, and data-dependent behavior) is how to set the model parameters. Unfortunately, there does not exist currently an extensive set of measurement data of real systems on which to base the parameterization, nor even a compelling folklore about what range of values are reasonable. We have therefore run a large number of experiments with parameters varying over a wide range. The results presented here represent a subset of those experiments that we believe fairly represents the "typical" behavior of the systems.

There are two kinds of parameters that must be given values: those involving time (T, L, and Q) and those involving size (P, J and K).

Considering first the parameters involving time, we have chosen to let the compute time T be the unit of time against which all other time parameters are measured; that is, we have set $T = 1$.

In all of our experiments we let the lock holding time L vary over an extensive range, in particular from 0.01 to 0.5. At the low end this represents extremely low lock contention, while at the high end it represents saturation of the lock.

We have run our experiments with a number of widely differing values for the scheduling quantum Q. Quantitatively, the results vary, sometimes significantly, with the value of Q. As might be expected, larger values of Q result in greater amounts of spinning. (Note that since we do not charge for context switches in our model, there is no performance penalty for smaller values of Q.) This is illustrated in Figure 1, which shows the mean number of processors spinning as a function of Q and the number of threads for a 5 processor system, in the lock contention situation. (The mean number of processors spinning has the advantage of being easily computed from our models, as well as being directly indicative of the mean spin time, and thus is shown in many of our graphs.) The increase in spinning with Q can be attributed to an increase in the variance of the spin time, which results from more occasional but longer lasting situations in which the thread being waited for (either the one holding the lock in the lock contention case illustrated in Figure 1, or the last one to reach the barrier in the case of barrier synchronization) is unscheduled. Note that this performance benefit for smaller scheduling quanta is quite distinct from the benefit in a sequential system of allowing the rapid completion of short jobs. While quantitatively our results depend on the specific value of Q chosen, the qualitative behavior is similar in all cases. We have chosen $Q = 1$ for all results presented here. This choice was in part motivated by the fact that the rate of increase in spin time with increasing values of Q (as illustrated in Figure 1 for the lock contention case) drops dramatically as Q increases beyond 1.

Turning our attention to the parameters involving size, we chose P and J by running a set of test experiments for the lock contention situation to determine how the behavior of the system depends on its size. In these experiments J was set equal to P and T was varied (rather than being fixed at 1 as it is elsewhere) so that the lock throughput (and thus the lock utilization) is nearly constant across all system sizes. (See Appendix B for details.) This allows us to isolate the changes in mean spin time caused by system size from those that would occur naturally because of increased lock contention in the larger systems if T were held invariant.

Figure 2a shows the mean number of processors spinning as a function of system size and mean lock holding time. Figure 2b is the same data normalized by the number of spinning processors in the 5

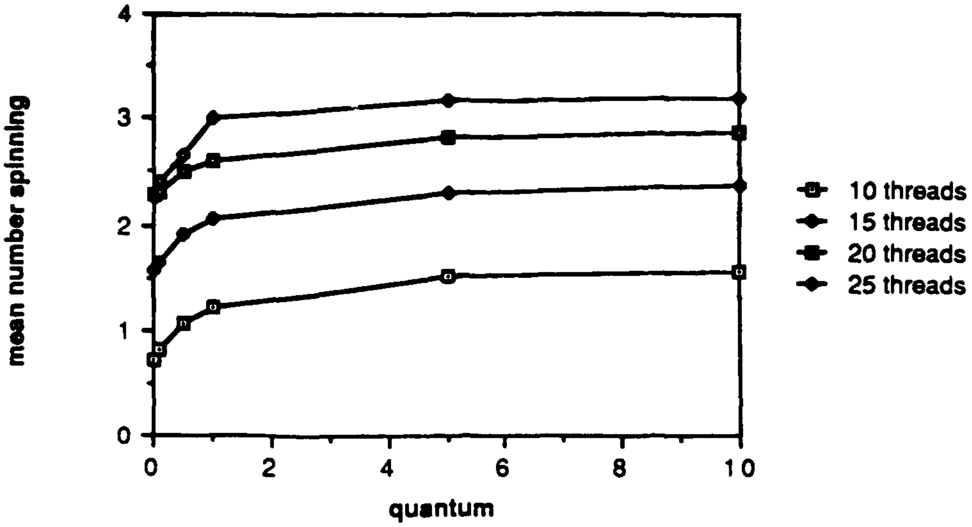


Fig. 1: Effect of Quantum Size; 5 Processor System

processor system. For short duration locks the behavior of the systems is nearly independent of system size (both in terms of the absolute difference in the spinning times, as shown in Figure 2a, and, somewhat surprisingly, in terms of the relative difference, as shown in Figure 2b). For long duration locks, on the other hand, the number spinning is nearly linearly proportional to system size. Based on this observation, we have chosen to present results for experiments with the two smallest system sizes, 5 and 10 processors, since these minimize the still considerable processing time required to run the experiments but still represent a factor of two difference in system size. Based on the above observations, results for larger systems for the lock contention situation can be safely extrapolated from those presented for the smaller systems.

Parameters P and J for the barrier synchronization case were chosen to be consistent with the lock contention results, that is, we again restricted P to 5 and 10 processors. In each model J is kept constant at the number of processors, since this lets us easily compare the baseline and multiprogramming cases. Finally, the number of other threads in the multiprogramming environment, K, was varied across an extensive range; we present selected results.

3. Results for Lock Contention

3.1. The Baseline Case

As noted previously, the baseline case represents the situation where the programmer has the greatest information available at implementation time about expected spin times. We assume that the parallel machine is dedicated to a single application during its execution and that the application has been partitioned to have precisely the same number of threads as there are processors. Thus, we model P processors and $J = P$ threads.

Performance measures for this baseline environment are used only for comparative purposes. The mean number spinning for this case is contained in the data given in Figure 3, in the context of a comparison with the multiprogramming case, which we discuss next.

3.2. Multiprogramming

We examine the effects of multiprogramming by comparing the mean number of spinning processors under multiprogramming (i.e., when the number of threads exceeds the number of processors) to that in the baseline case. We have run our experiments with J varying from P to $5P$. The performance results obtained are qualitatively similar, so we have extracted the results for $J = 2P$ for presentation here.

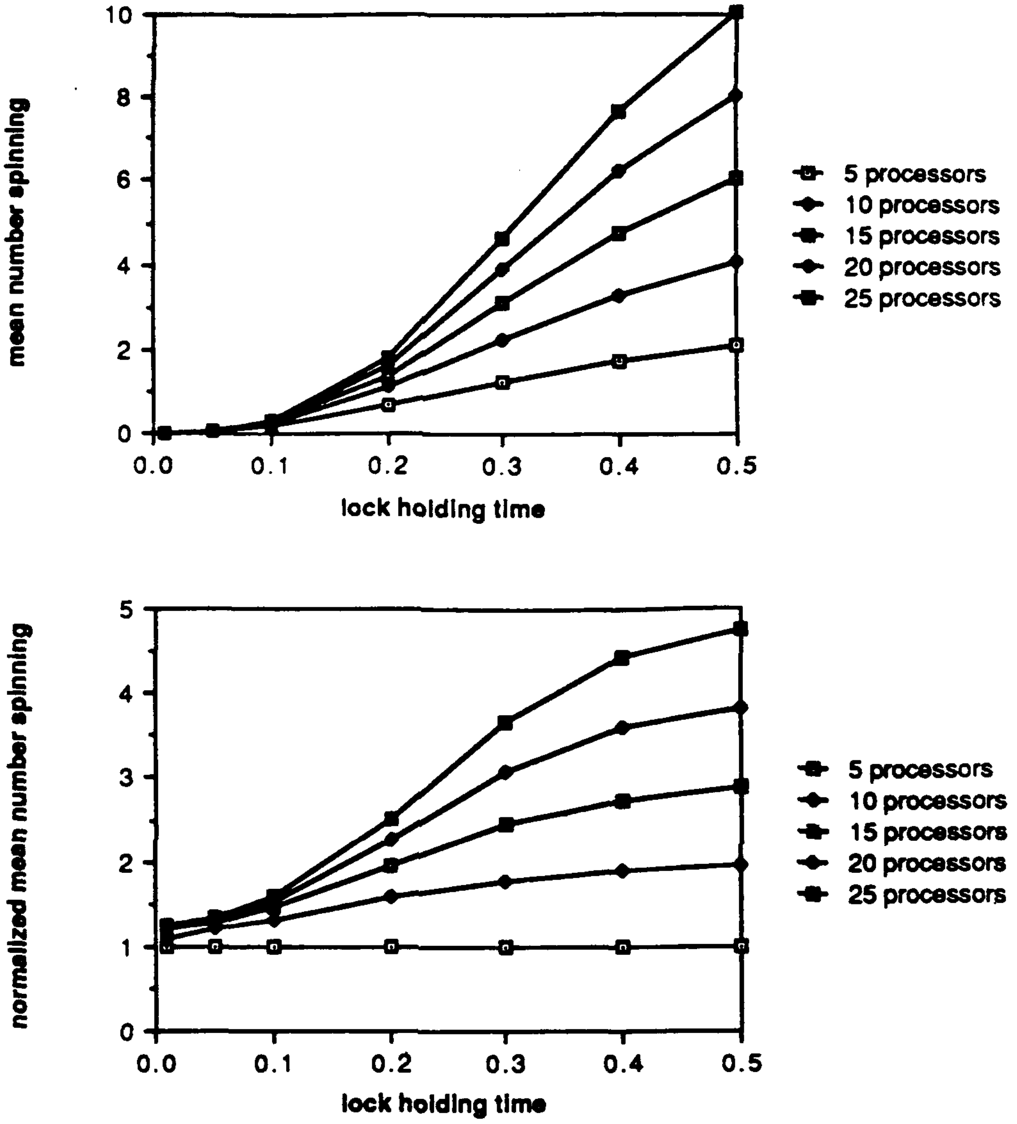


Fig. 2a/2b: Effect of System Size

It might appear at first that doubling the number of threads would result in increased lock contention, and that any increase in the number of spinning processors would be a combination of this effect and the effect of multiprogramming. As noted previously, though, lock contention is inherently dependent on the number of processors, not the number of threads. Because the number of processors is kept constant, the total instruction delivery rate, and so the total rate of lock requests and the resulting lock contention, also are kept constant, with the exception of changes due solely to differing patterns of thread executions. Thus, any change in system performance can be attributed solely to the introduction of multiprogramming.

Figure 3 presents a comparison of the baseline and multiprogramming environments for 5 and 10 processor systems. When lock contention is low (represented here by short lock holding times) system performance is not significantly affected by multiprogramming. Thus, in these environments the author of parallel software can choose between spinning and blocking as though the application were to be run standalone. However, at modest to high lock contention, multiprogramming causes a significant degradation in performance. This effect is the result of the fact that the thread holding the lock is occasionally unscheduled. In these instances other threads will spin for a scheduling quantum, a considerable period of time.

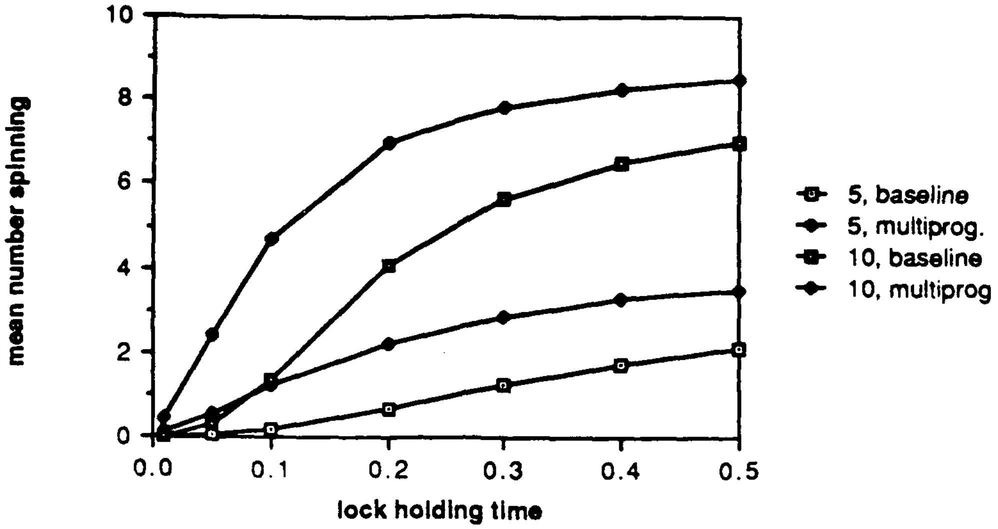


Fig. 3: Baseline Case and Multiprogramming; 5 and 10 Processor Systems

The above analysis has assumed that the scheduling discipline is oblivious to the internal behavior of the application software, that is, to the states of the threads involved in the scheduling decision. It might be possible to obtain better system performance if the scheduler had access to this information, since, for example, the scheduler could then avoid unscheduling the thread that holds the lock. To accomplish this, the scheduler must rely on the application software (perhaps through the code implementing the language primitives supporting locking) to set a flag when it acquires a lock, because (for efficiency reasons) the operating system is not involved in lock requests. However, this might have one or both of the following undesirable effects. First of all, if it required that an action on the flag be performed inside the critical section, it would increase by at least one instruction the lock holding time, which could be critical to system performance [Dritz & Boyle 1987]. Second, an unscrupulous user might be able to modify his code so as to set this flag for all his threads in an attempt to obtain better service [Coffman & Kleinrock 1968].

We have investigated the potential performance benefits that could be obtained if the scheduler had knowledge of the state of the threads, i.e., whether they were computing, spinning, or holding the lock. We have investigated three policies that use this information. In Discipline A the scheduler never unschedules a thread holding the lock. This eliminates the situation where threads are spinning uselessly waiting for the lock to be released by an unscheduled thread. Discipline B allows the thread holding the lock to be unscheduled, but will not schedule a currently unscheduled spinning thread unless the lock is free. This discipline has the same goal as the first, to reduce useless spinning, but it reduces the motivation of a user to lie about the state of his threads. The final policy, Discipline C, combines both of the previous modifications.

Figure 4 presents a summary of the effects of these improved scheduling policies on system performance. As is readily seen, all three policies result in significant improvements in system performance, and nearly eradicate the performance penalty imposed by multiprogramming (cf. Figure 3). Discipline A is preferable to Discipline B, and yields performance almost as good as when both modifications are combined.

Perhaps one of the less intuitive characteristics of Figure 4 is the shape of the curve for Discipline B. For low lock holding times (less than about 0.1 in Figure 4b, for example), Discipline B yields significantly worse performance than that of Disciplines A and C. The curve then exhibits a distinct change in shape (at around a lock holding time of 0.1 in Figure 4b), and for larger lock holding times quickly converges to the curves for Disciplines A and C. This shape can be (at least partially) explained as follows. For low lock holding times, there are usually no or very few threads (either with or without processors) that are in the spinning state. Thus, Discipline B yields little improvement in performance in this case. As the lock holding time increases, the average number of threads in the spinning state

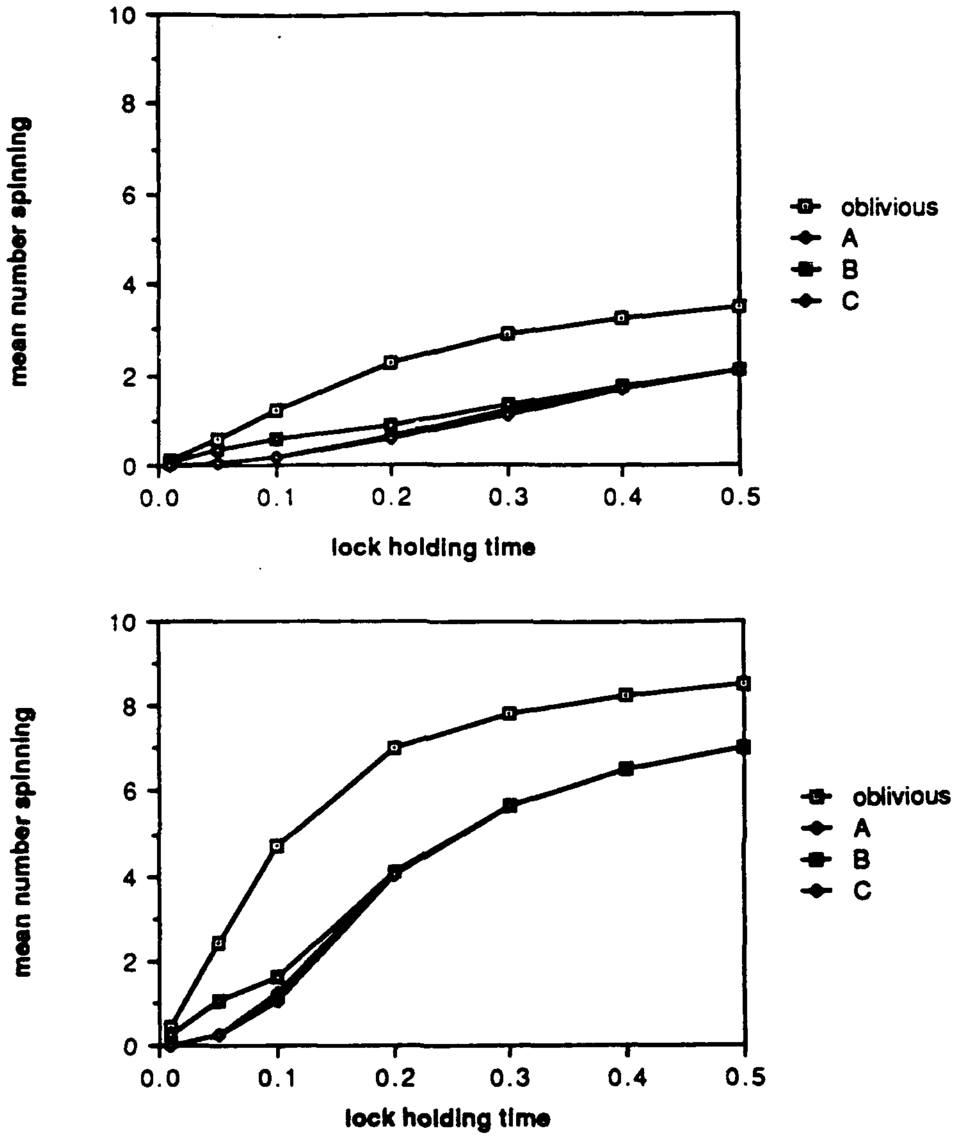


Fig. 4a/4b: Effect of Scheduling Policy; 5 and 10 Processors (cf. Fig. 3)

increases. With Discipline B, such threads tend to be kept unscheduled, diminishing the contention for processors among the remaining threads. Therefore, although the lock holder flmay be context-switched in discipline B, this becomes increasingly less likely (because of the absence of a suitable thread to switch it with), and in any case the average time until that lock holder is rescheduled becomes very small, as the lock holding time is increased. Thus, for large lock holding times, Discipline B closely corresponds to Disciplines A and C.

All three improved policies also render the system nearly insensitive to the total number of threads in terms of the mean number of processors spinning. Figure 5 illustrates this effect using Discipline A as an example.

It is natural to ask in the multiprogramming context which of spinning and blocking is the preferable waiting mechanism. To at least partially address this question, we have chosen to give an informal threshold for context switch times as a function of the parameters of our model. This threshold is such that context switch times less than the specified value should result in blocking being preferable to spinning (within the assumptions of the model). Context switch times greater than the threshold value *may* still result in blocking being preferable, although for context switch times much larger than the threshold this is unlikely. Note that, in practice, context switch times depend heavily on the specific

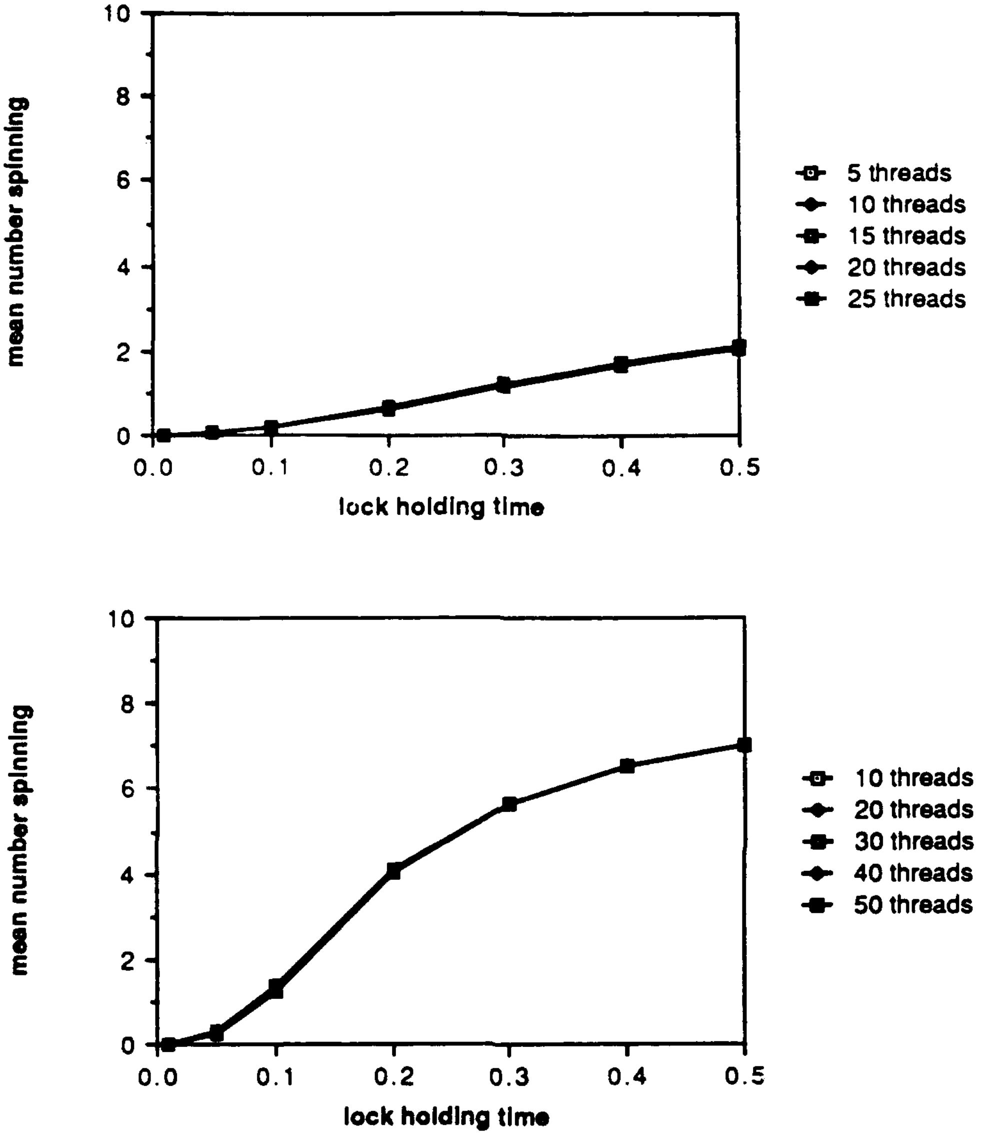


Fig. 5a/5b: Discipline A Performance vs. Number of Threads; 5 and 10 Processors

architecture and level of granularity of parallelism [Polychronopoulos & Kuck 1987]. For this reason, and due to the abstract nature of our model, the threshold values we obtain are more useful as relative values (when comparing the various scheduling disciplines) than as absolute values.

The threshold is computed using an approximation to the average spin time per initially unsuccessful lock request, considering only those time intervals during which at least one unscheduled thread is in the compute or critical section states.[2] We ignore spin time when there are no unscheduled threads in these states because blocking is not useful in that case. Blocking is likely to be advantageous if the context switch time is smaller than the threshold value, since blocking should result in a smaller amount of wasted processor time in this case.

---

[2] The approximation assumes that the rate of initially unsuccessful lock requests is independent of the presence or absence of unscheduled non-spinning threads. Perhaps surprisingly, we found that the nature of the results was insensitive not only to the use of this approximation (rather than an exact analysis), but also to the precise way in which the threshold was defined (alternate definitions gave equivalent results).

Figures 6a and 6b present the threshold values for the four multiprogramming scheduling disciplines in systems with 5 and 10 processors and twice the number of threads as processors. (The thresholds for larger numbers of threads mimic the shape of the values given, although of course are larger in value. The thresholds for the "oblivious" scheduling discipline and Discipline B are relatively sensitive to the number of threads in the system, while those for Disciplines A and C are largely unaffected by that parameter.) The threshold values decline for large lock holding times for some of the disciplines because of the decreasing probability that any of the unscheduled threads have useful work to do.

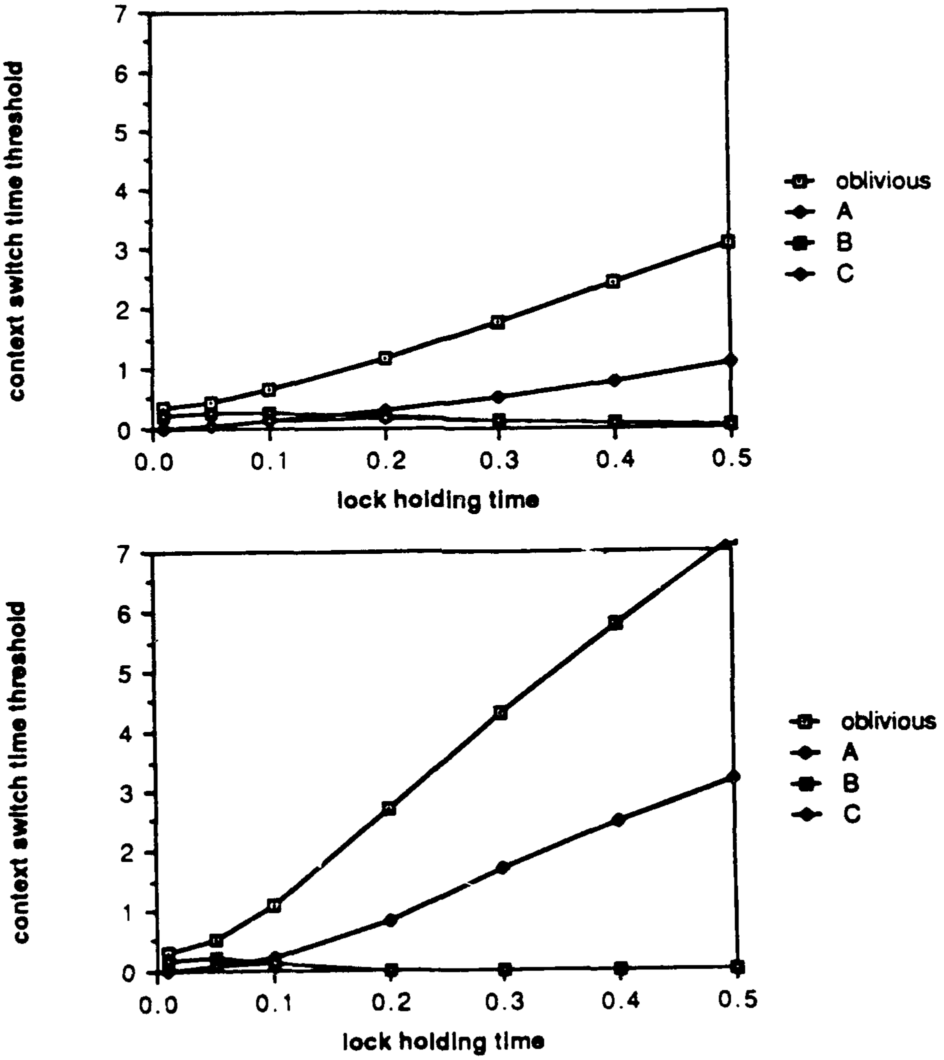


Fig. 6a/6b: Context Switch Time Thresholds; 5 and 10 Processors

It is clear from Figure 6 that there is a significant qualitative difference in the behavior of the various multiprogramming scheduling disciplines. In particular, the disciplines that refuse to schedule a spinning thread unless the lock is free (Disciplines B and C) greatly reduce the range of lock utilizations over which blocking may be an appropriate waiting mechanism. This is a characteristic that argues in favor of scheduling disciplines that embody this feature.

3.3. Data-Dependent Behavior

The final environment considered is that of data-dependent behavior. Recall that this is reflected in our model by a parameter p. A thread acquiring the lock releases it in zero time with probability $1-p$, and with probability p holds it for mean time $\frac{L}{p}$. The case $p = 1.0$ corresponds exactly to the baseline case. Figure 7 gives the ratio of the mean number of spinning processors for various values of p to the mean number spinning when $p = 1.0$. We make three observations based on this data. First, variability has the greatest effect when lock contention is low. In these cases the total amount of spinning is small in any case, so despite the potentially large percentage increase the difference in spinning is small in absolute terms. (The absolute amount of spinning occurring for the baseline case of p=1.0 can be found in Figure 2 for the 5 and 10 processor systems.) Second, quite high variability is required before any significant effect is observed. In our data a p of 0.5 is required before even a factor of two difference occurs. Finally, the behavior of the system as a function of data-dependence is nearly identical in the 5 and 10 processor systems. Thus, we conclude that the effect of data-dependent behavior is roughly independent of system size.

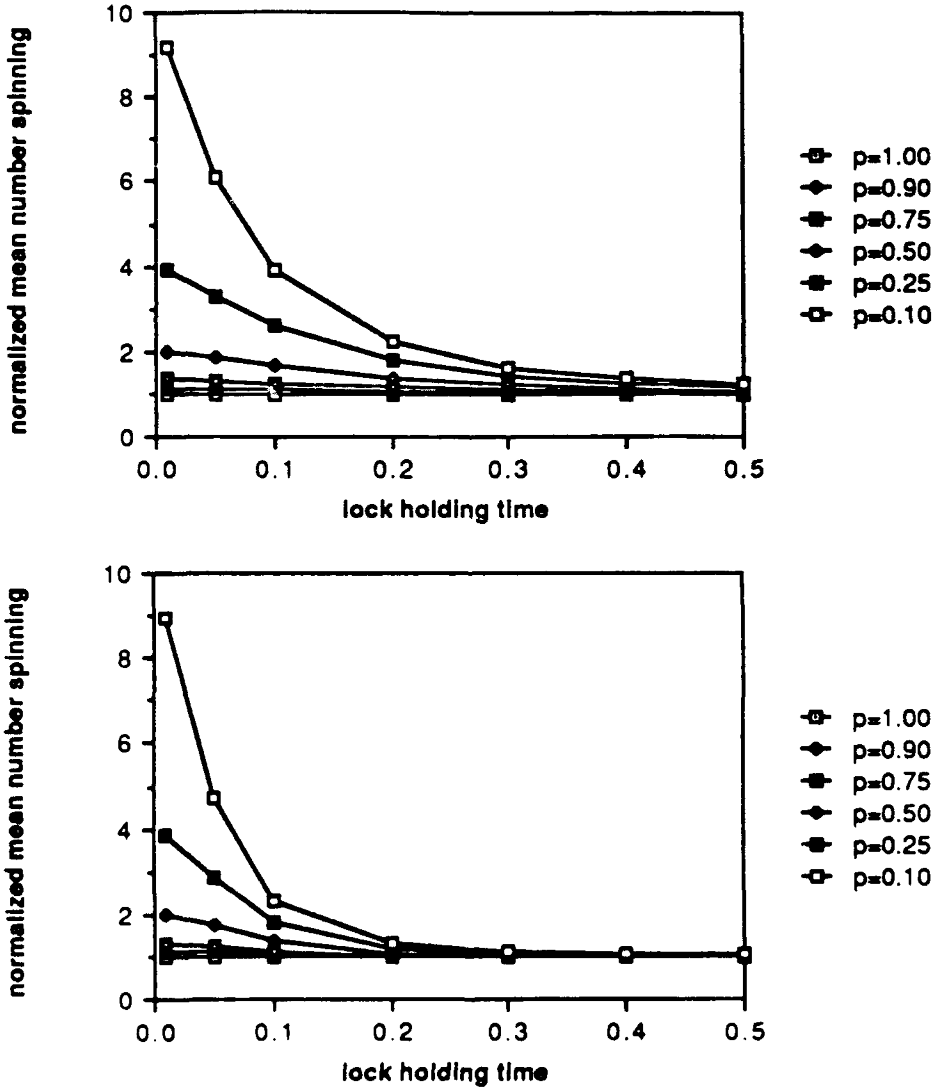


Fig. 7a/7b: Data-Dependent Behavior; 5 and 10 Processors

4. Barrier Synchronization

4.1. The Baseline Case

We now turn our attention from lock contention to barrier synchronization. Figure 8 shows how the amount of spinning is affected by the number of threads involved in the barrier synchronization. Here we have assumed that $J = P$, that is, that all threads have a processor dedicated to them. Under the assumptions of our model it is easily shown that the mean time to complete the barrier is $\sum_{i=1}^{J} \frac{1}{i}$.

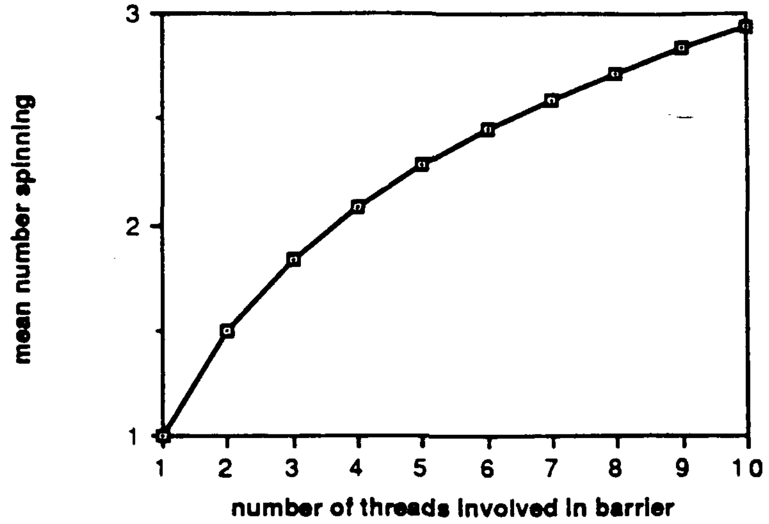


Fig. 8: Barrier Synchronization Baseline Case

4.2. Multiprogramming

To reflect multiprogramming, the number of threads involved in the barrier synchronization is kept constant at P, but K other threads are introduced that compete for processors. Figure 9a illustrates how the mean number of spinning processors is affected by the number of these "other threads" as a function of system size. Clearly, the amount of spinning per processor per time unit decreases with additional other threads because those threads never spin. Figure 9b shows how the amount of spinning per barrier thread per barrier synchronization increases with the amount of competition for processors. It gives the mean time required to achieve the barrier synchronization for various system sizes. There is a nearly linear relationship between the number of other threads and the mean time to achieve the barrier.

Just as in the lock contention situation, it is natural to ask if system performance can be improved by giving the scheduler some information about the internal state of the threads. For barrier contention, the only information that seems useful is which threads are spinning. Then if a spinning thread happens to be descheduled because its quantum has expired, it would seem to be beneficial not to reconsider scheduling it again until the barrier had been reached by all other processors.

Figure 10 shows the ratio of the performance under this modified discipline to the performance achieved under the "oblivious" discipline. It is not terribly surprising that the mean number of spinning processors is reduced by this modification. What is surprising is that this gain in overall system performance does not penalize the threads involved in the barrier. The explanation for this is that spinning threads compete with those threads still working toward the barrier. Thus, a mechanism that tends to eliminate the spinning threads helps the other threads achieve the barrier. This effect evidently outweighs the disadvantage that under the modified discipline many more of the barrier threads are unscheduled at the time the last thread reaches the barrier, and so at the time the threads begin working toward the next barrier, than under the "oblivious" discipline. Note though, that under the modified discipline, as with the oblivious discipline, the performance of the barrier threads still degrades considerably with increasing numbers of other threads.

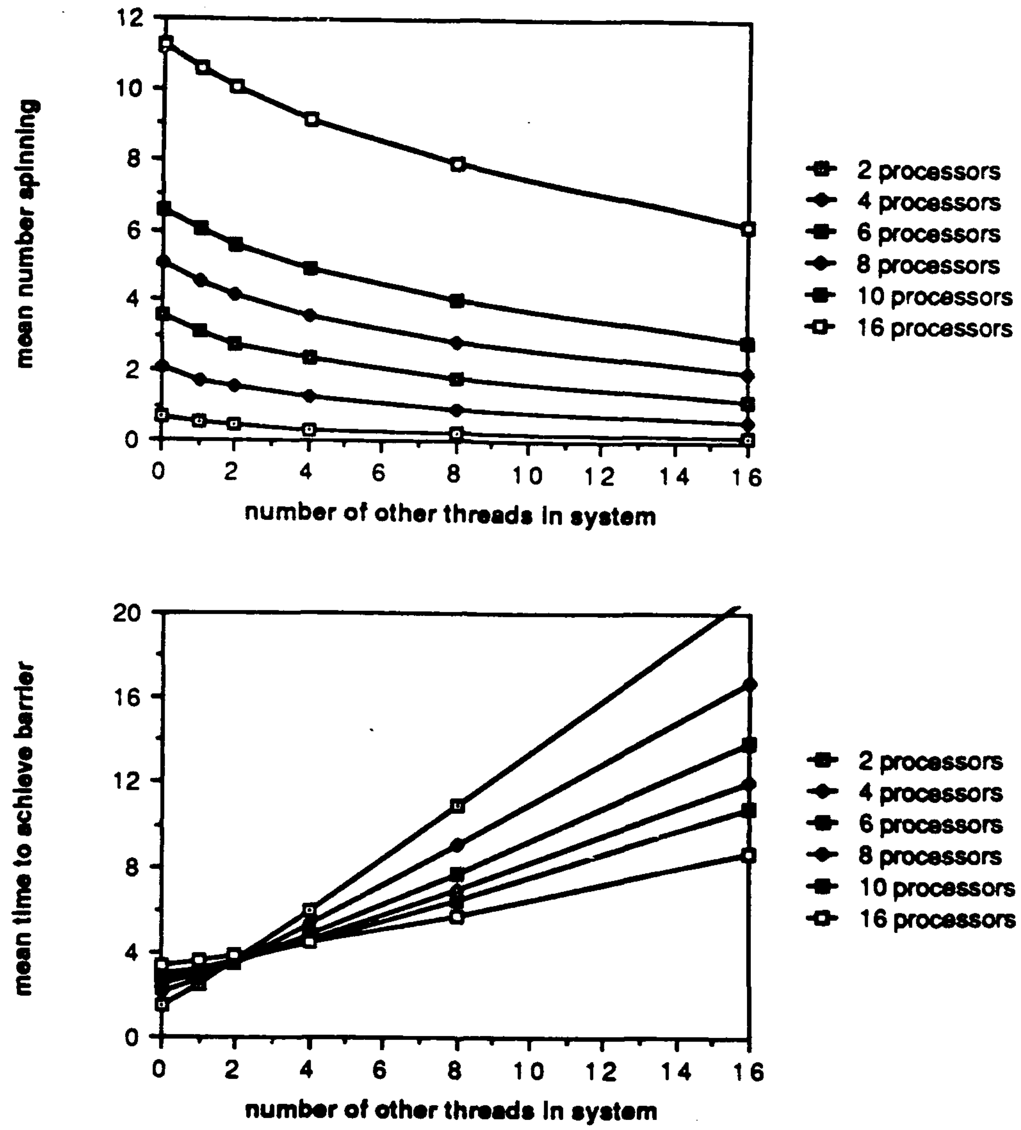


Fig. 9a/9b: Barrier Synchronization under Multiprogramming

4.3. Data-Dependent Behavior

The threads involved in the barrier synchronization in the baseline case were "balanced", in the sense that the amount of service each required before reaching the barrier was chosen from a single distribution. To model data-dependent behavior we examine the effect of introducing imbalance. We do this by letting some threads reach the barrier in zero time, while other threads take longer than the overall mean time. Recall that parameter p is the probability that a thread requires a non-zero service time.

Figure 11 shows how the amount of spinning is affected by the uncertainty in the thread execution times. For each value of p, we have graphed the ratio of the fraction of time each processor spends doing useful work against that value when $p = 1.0$. It is clear that variance can have a substantial effect on the expected spin times of threads using barrier synchronization. Further, the magnitude of this effect increases with the size of the parallel machine. Since we have experimented with quite modest system sizes, one would expect that in real systems data-dependent behavior could be quite significant to the performance of applications using barrier synchronization.

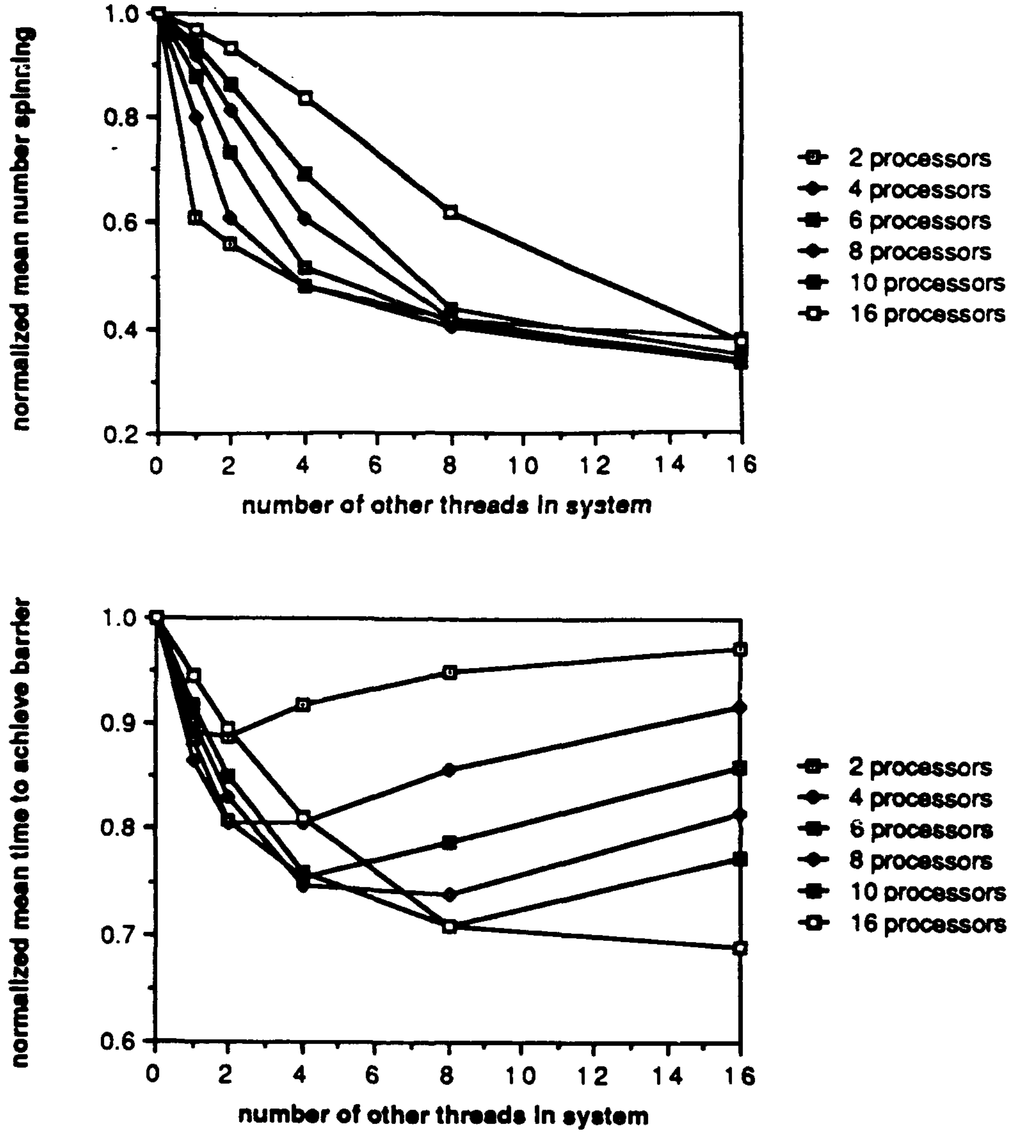


Fig. 10a/10b: Normalized Performance of Modified Discipline (cf. Fig. 9)

5. Conclusions

We have used two analytic models to compare the amount of spinning that occurs in various environments when threads either compete to obtain a lock or synchronize at a barrier. The purpose of our comparison is to determine if the uncertainty in performance caused by multiprogramming or by data-dependent behavior significantly increases the amount of spin time that occurs, and so complicates the task of choosing an appropriate waiting mechanism.

We have found that for lock contention, neither source of uncertainty poses much danger, assuming that the system scheduler has access to information concerning who holds the lock or who is spinning. However, for barrier synchronization, the amount of spinning is quite sensitive to these forms of uncertainty. Thus, to correctly choose a waiting mechanism the programmer requires fairly precise information about not only the behavior of his program but also about the load that will be placed on the machine when his application is run. For this case, then, the programmer's task is considerably more complicated in multiprogramming and/or data-dependent environments than in the case of the more controlled environment of a dedicated machine and predictable running times.

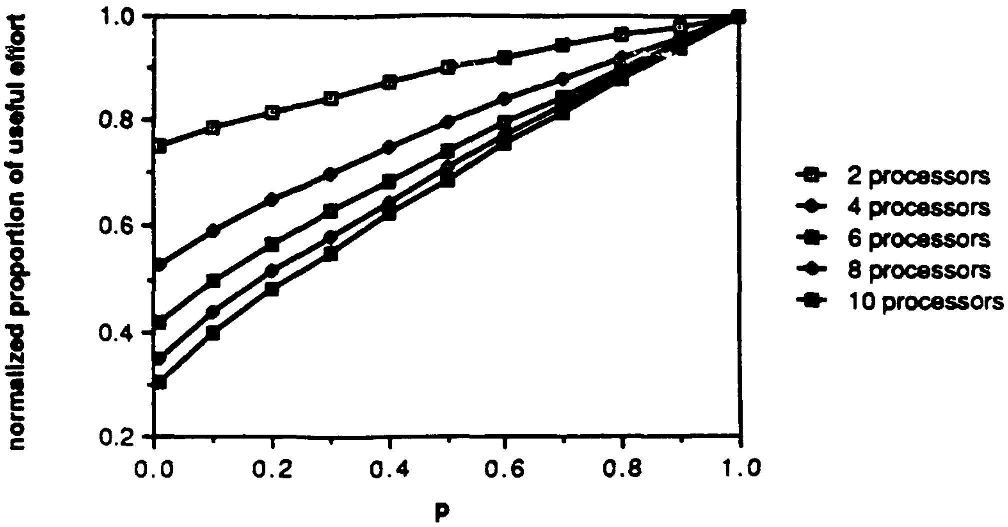


Fig. 11: Normalized Data-Dependent Behavior

Acknowledgements

Partial support for this work was generously provided by Bell Communications Research, Boeing Computer Services, Digital Equipment Corporation, Tektronix, Inc., the Xerox Corporation, and the Weyerhauser Company. The Centre National de la Recherche Scientifique, France, and Laboratoire MASI, University of Paris 6, provided generous support and resources for Zahorjan for the year sabbatical leave during which this work was performed.

References

[Beck et al. 1987]
B. Beck, B. Kasten, and S. Thakkar. VLSI Assist for a Multiprocessor. *Proc. 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1987), pp. 10-20.

[Coffman & Kleinrock 1968]
Edward G. Coffman, Jr., and Leonard Kleinrock. Computer Scheduling Methods and their Countermeasures. *Proc. 1968 Spring Joint Computer Conference*, pp. 11-21.

[Dritz & Boyle 1987]
Kenneth W. Dritz and James M. Boyle. Beyond "Speedup": Performance Analysis of Parallel Programs. Technical Report ANL-87-7, Mathematics and Computer Science Division, Argonne National Laboratory, February 1987.

[Dubois & Briggs 1982]
M. Dubois and F.A. Briggs. An Approximate Analytical Model for Asynchronous Processes in Multiprocessors. *Proc. 1982 International Conference on Parallel Processing*, pp. 290-297.

[Eager et al. 1988]
D.L. Eager, E.D. Lazowska, and J. Zahorjan. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. *Proc. 1988 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, May 1988.

[Kleinrock 1975]
L. Kleinrock. *Queueing Systems: Volume I: Theory*. John Wiley and Sons, 1975.

[Lazowska et al. 1984]
E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[Lipsky & Church 1977]
L. Lipsky and J.D. Church. Applications of a Queueing Network Model for a Computer System. *Computing Surveys 9*,3 (September 1977), pp. 205-222.

[Lo & Gligor 1987]
S.-P. Lo and V.D. Gligor. A Comparative Analysis of Multiprocessor Scheduling Algorithms. *Proc. 7th International Conference on Distributed Computing Systems* (September 1987), pp. 356-363.

[Ousterhout 1982]
John K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proc. 3rd International Conference on Distributed Computing Systems* (October 1982), pp. 22-30.

[Polychronopoulos & Kuck 1987]
C.D. Polychronopoulos and D.J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers C-36*,12 (December 1987), pp. 1425-1439.

[Rodgers 1986]
David P. Rodgers. Personal Communication. October 1986.

[Stewart 1978]
W.J. Stewart. A Comparison of Numerical Techniques in Markov Modelling. *CACM 21*,2 (February 1978), pp. 144-152.

Appendix A: Details of the Analysis of the Models

In this appendix we specify more precisely the models we have used and the analysis by which we have obtained the results presented in the body of this paper. We describe only the model for the lock acquisition situation. The model for barrier synchronization is similar.

We use a Markovian model to represent a system with P processors and $J \geq P$ threads. A state of the system is given by a six-tuple $(n_1,n_2,n_3 | n_4,n_5,n_6)$. Here n_1, n_2, and n_3 are the number of threads currently scheduled (that is, allocated processors) that hold the lock, are spinning, and are computing respectively, and n_4, n_5 and n_6 are the corresponding counts of unscheduled threads in those three states. Thus, there are a total of $2P(J-P)+2J-P+1$ states, of which $J-P+1$ are of the form $(0,0,P | 0,n_5,J-P-n_5)$, $P(J-P+1)$ are of the form $(1,n_2,P-n_2-1 | 0,n_5,J-P-n_5)$, and $(P+1)(J-P)$ are of the form $(0,n_2,P-n_2 | 1,n_5,J-P-n_5-1)$.

A computing thread makes a lock acquisition attempt after an amount of service (i.e., time on a processor) exponentially distributed with mean T. With probability p, a thread acquiring the lock releases it after an exponential amount of service exponentially distributed with mean $\frac{L}{p}$. With probability $1-p$ the lock is released in zero time. This is actually modelled by having "multi-step" transitions in the Markov model, that is, transitions between states that imply the movement of more than a single customer. Details on this follow when the state transition rates are defined.

Here only the "oblivious" multiprogramming scheduling discipline is considered. (The modifications required for the other scheduling disciplines are straightforward.) Each thread allocated a processor is descheduled after an amount of time exponentially distributed with mean Q. A currently descheduled thread is chosen at random as a replacement.

The steady state solution of this model is obtained by solving the state flow balance equations [Kleinrock 1975]. It is difficult to give the flow balance equations in a compact form. We therefore write down the same information in a different form, giving simply the rate of flow out of each state and the state to which that flow enters.

Let $S \equiv (n_1,n_2,n_3 \mid n_4,n_5,n_6)$ be a state of the system. Let S_i be obtained from S by subtacting 1 from n_i, S^i be obtained by adding 1 to n_i, and allow these operations to be applied repeatedly. For example, $S_{3,5}^{2,6} = (n_1,n_2+1,n_3-1 \mid n_4,n_5-1,n_6+1)$. Then the flow out of state S is given by:

$$S \Longrightarrow S_3^2 \text{ with rate } (n_1+n_4)\frac{n_3}{T}$$

$$S \Longrightarrow S_3^1 \text{ with rate } (1-n_1-n_4)\frac{n_3}{T}p$$

$$S \Longrightarrow (1,n_2-i,n_3+i \mid n_4,n_5,n_6) \text{ with rate } \frac{n_1p}{L}(1-p)^{i-1}p \quad , \quad 1 \le i < n_2$$

$$S \Longrightarrow (0,0,P \mid n_4,n_5,n_6) \text{ with rate } \frac{n_1p}{L}(1-p)^{n_2}$$

and, for $J > P$,

$$S \Longrightarrow S^{i,3+j,3+i} \text{ with rate } \frac{n_j}{Q}\frac{n_{3+i}}{J-P}, \qquad 1 \le i \ne j \le 3$$

We obtained the solution of the flow balance equations by the power method [Stewart 1978]. This is an asymptotically exact iterative technique involving repeated multiplication of the current estimate of the steady state probability vector with the transition matrix. We initialized the probability vector so that all states had equal probabilities. We stopped the iteration when the sum of the absolute values of the changes in the state probabilities in successive iterations was below a threshold of 0.00005. We use the sum of the changes in all states rather than the maximum change in any one state because we found that this latter measure lead to unreliable results. Our threshold value was determined to be adequate by solving a number of test cases starting with relatively large thresholds and then repeatedly halving the threshold and resolving. Comparing the results obtained each time the threshold was halved, we informally concluded that the results were reliable when halving the threshold did not produce an appreciable change. We know of no problems with the accuracy of the solutions we have obtained using this threshold, but we did observe that some models required a very large number of iterations to reach convergence. This was typically the case when there were large differences in time scales in the model. For instance, we ran some cases with T=1, Q=1 and L=0.001. These models often resulted in long execution times.

Given the steady state probability vector provided by the power method, computing performance measures is straightforward. For instance, denoting the steady state probability of state S by $P(S)$, the mean number of spinning processors is given by

$$\sum_{\text{all states } S} n_2 P(S)$$

and the lock throughput rate is given by

$$\sum_{\text{all states } S} \frac{n_1}{L} P(S)$$

One reservation that might be raised about our model is that all service time distributions are exponential. Indeed, the work by Dubois and Briggs [1982] presents a more complicated (and more restrictive) model requiring a heuristic analysis with the sole purpose that lock holding times can be made less variable than the exponential. We believe that exponentials are acceptable in our models for two reasons. First, as explained in the introduction, our results depend on comparing models all of which use exponentials. Experience shows that in general these sorts of comparisons are highly robust to inaccuracies such as the choice of service time distribution [Lipsky & Church 1977, Lazowska et al. 1984]. Second, in a similar model in the domain of load sharing [Eager et al. 1988] the specific effect of the exponential distribution was explored and compared against results obtained from a deterministic distribution. In that work, which was also based on the comparison of models, very little difference was observed between the exponential and the deterministic set of models.

Appendix B: Estimating Compute Times

In the set of test experiments that were ran to determine the effect of system size on its behavior (described in Section 2.3), T was varied so that the lock throughput was nearly constant across all system sizes. In this appendix, we briefly describe the manner in which this was done.

Because lock throughput depends on T, choosing the T required to exactly equalize lock throughputs across system sizes requires iteration and its consequent high cost. We therefore chose to use a simpler but effective approximate technique that does not require iteration. This approximation is obtained by assuming that no lock contention will take place. Under this assumption lock throughput is given by $\frac{P}{T+L}$, and so lock utilization is $\frac{PL}{T+L}$.

The procedure we followed was to solve the model with the chosen values of L and T for the 5 processor system and to note the resulting lock utilization U. For a P processor system we then solved for the appropriate compute time T_P as

$$U = \frac{PL}{T_P + L}$$

Our assumption of no contention is clearly valid for low values of lock holding time, but it is not clear how well it performs for larger values. We therefore monitored the lock throughput actually resulting from our choices for the T_P. In no case did the lock throughput rate differ from that of the 5 processor system by more than 10%.

The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors

Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy

Department of Computer Science
University of Washington
Seattle WA 98195

Abstract

Threads ("lightweight" processes) have become a common element of new languages and operating systems. This paper examines the performance implications of several data structure and algorithm alternatives for thread management in shared-memory multiprocessors. Both experimental measurements and analytical model projections are presented.

For applications with fine-grained parallelism, small differences in thread management are shown to have significant performance impact, often posing a tradeoff between throughput and latency. Per-processor data structures can be used to improve throughput, and in some circumstances to avoid locking, improving latency as well.

The method used by processors to queue for locks is also shown to affect performance significantly. Normal methods of critical resource waiting can substantially degrade performance with moderate numbers of waiting processors. We present an Ethernet-style backoff algorithm that largely eliminates this effect.

1. Introduction

The purpose of this paper is to study the performance implications of thread management alternatives for shared-memory multiprocessors.

In traditional operating systems, a process, consisting of a single address space and a single thread of control within that address space, is used to execute a program. Within the process, program execution entails initializing and maintaining a great deal of state information. Page tables, swap images, file descriptors, outstanding I/O requests, and saved register values are all kept on a per-program, and thus per-process, basis. The sheer volume of this information makes processes expensive to create and maintain.

Threads, or "lightweight" processes, separate the notion of execution from the rest of the definition of a process. A single thread executes a portion of a program, cooperating with other threads concurrently executing within the same address space. Like processes, every thread must have a separate program counter and stack of activation records, describing the state of its execution. However, much of what is normally kept on a per-process basis can be maintained in common for all threads executing in a single program, with dramatic reductions in overhead.

Thread packages have become a common element of new languages and operating systems for both uniprocessor and multiprocessor architectures. Mach [Accetta et al. 1986], Topaz [Thacker et al. 1988], Psyche [Scott et al. 1988], DYNIX [Sequent 1988], and several extensions to UNIX [Bach & Buroff 1984; Edler et al. 1988] are examples of operating systems that provide explicit support for concurrent or parallel execution of programs. Ada [Mundie & Fisher 1985], CSP [Hoare 1978], PRESTO [Bershad et al. 1988a], Mesa [Lampson & Redell 1980], Concurrent Euclid [Holt 1982], and Emerald [Jul et al. 1988] evidence equal interest within the language community.

On uniprocessors, threads are used as a program structuring aid or to overlap I/O with processing. The metric of goodness for these thread management implementations is simply processing cost per thread creation or context switch. No locking is needed inside thread routines, since only one routine can be executing at any one time.

Programs on multiprocessors use threads to exploit parallelism. The speedup achievable by any given application depends on the availability of thread management routines that provide low cost facilities that are not a serial bottleneck. In Sequent's DYNIX operating system, for example, applications must use normal UNIX-like processes for parallelism [Sequent 1988]. Since process creation in DYNIX takes over 25 milliseconds, only very coarse-grained parallelism can be exploited. As another example, the Topaz kernel provides relatively inexpensive thread creation and synchronization, but the routines are protected by a single lock [Thacker et al. 1988]. While this may be appropriate for architectures with small numbers of processors, as the number of processors increases, the single lock could limit speedups for applications with fine-grained parallelism.

Our initial experience in the area of high-performance thread packages was with PRESTO, an application-level runtime library that relies on the kernel only for processor allocation and memory management [Bershad et al. 1988a]. This work showed that there is an order of magnitude performance advantage to using threads instead of DYNIX processes for exploiting parallelism. Drawing on this experience, we implemented a thread package that is, in turn, an order of magnitude faster than PRESTO. This basic package was then modified to implement each alternative we wanted to explore.

One consequence of the speed of our basic thread package is that small changes in the organization of data structures and locks have a significant impact on performance. Often, the choice involves a tradeoff between latency and throughput. Per-processor data structures can sometimes be used to avoid locking, however, improving latency and throughput at the same time.

This material is based on work supported by the National Science Foundation (Grants No. CCR-8619663, CCR-8703049, and CCR-8700106), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1989 ACM 0-89791-315-9/89/0005/0049 $1.50

Another consequence of the speed of our thread package is that its performance depends noticeably on the algorithm used to queue for locks. Earlier, we studied the relative performance of spinning and blocking locks [Zahorjan et al. 1988]. In general, a thread that tries to acquire a lock that is already held can either spin ("busy-wait") until the lock is released, or relinquish the processor. However, within the thread management routines themselves, spinning is the only option. Thus, blocking at user level may require spinning in thread management routines. Spinning has a cost both to the processor awaiting the lock and to processors doing useful work. The degradation of other processors becomes substantial for moderate numbers of waiting processors, especially for small critical sections. We present an Ethernet-style backoff algorithm that largely eliminates this effect.

The following sections describe these issues in more detail. In Section 2 we present an abstraction of a thread package: its objects, resources, and operations. Section 3 outlines the strategies for thread management that we examined and presents measurements of their relative performance. Section 4 compares methods of queueing for locks. Section 5 combines these results in an analytical model. Section 6 summarizes our experiences.

2. An Abstract Thread Package

As noted in Section 1, threads gain efficiency by separating the notion of execution from the rest of the definition of a process. The data structures needed by each thread are a program counter, a stack, and a control block. (The control block contains state information needed for thread management. Through the control block, the thread can be put onto lists and other threads can synchronize with it.) Another important data structure is the ready queue, which lists threads that are ready to run. Lampson and Redell [1980] provide a good description of the functionality of a uniprocessor thread package.

Thread operations are shown in Table 2.1. Creating a thread can be viewed as calling a procedure, except that the callee can execute in parallel with the caller. In both cases, the caller specifies a place to begin executing and some number of arguments. In fact, thread creation and startup is semantically equivalent to an asynchronous procedure call.

Thread Creation
    Allocate and initialize a control block, saving the initial PC.
    Allocate a stack and copy in the thread's arguments.
    Place new thread on the ready queue.

Thread Startup
    Remove thread from ready queue and begin to execute it.

Thread Block (wait on blocking lock, condition variable, or message)
    Save register values and PC on the thread's stack.
    Place thread on the condition queue for the event.
    Look for a thread in the ready queue, and start or resume it.

Signal a Blocked Thread
    Remove thread from the condition queue.
    Place the thread on the ready queue.

Thread Resume
    Remove thread from the ready queue.
    Restore registers.
    Continue executing it from the saved PC.

Thread Finish
    Deallocate the stack and control block.
    Look for a thread in the ready queue, and start or resume it.

Table 2.1: Thread operations

As Table 2.1 shows, a program can create a thread even if there is no idle processor available to run it. Because the parallelism cannot be immediately exploited in this case, it might seem that the overhead of thread creation should be avoided. The program may run faster by creating the thread, however, if at some future time there will be an idle processor that can be used to execute the thread. This idea of creating parallelism for future use is very powerful. Unfortunately, in the above framework, its space cost is prohibitive. Each thread must be initially allocated a large amount of space for its stack, since it is expensive to dynamically expand the space if the thread later runs out of it. In Table 2.1, the thread is allocated space for a stack when it is created, but the space is largely wasted until the thread is actually started. Using virtual memory could remove the need to allocate physical memory to back the stack space until the thread begins to run; however, allocating extra virtual memory also is expensive.

An important optimization to Table 2.1, therefore, is to copy a thread's arguments into its control block when the thread is created. This way, the stack need not be allocated until thread startup; the arguments can be copied from the control block to the stack at that time. WorkCrews [Vandevoorde & Roberts 1988] and PRESTO [Bershad et al. 1988a] both take this approach.

Another important optimization is to store deallocated control blocks and stacks in free lists [Bershad et al. 1988a]. If these data structures were individually allocated out of the heap, thread overhead would include the cost of finding a free block of the correct size as well as possibly coalescing the block when it is returned to the heap. By using free lists, both allocation and deallocation can normally be simple list operations.

We begin our study by assuming these optimizations. For simplicity, we will focus on the effect of thread management alternatives on the performance of only a few thread operations: creation, startup, and finish. These operations manipulate each of the three shared data structures: the ready queue, the stack free list, and the control block free list. Most of the discussion applies as well to threads that block and resume.

3. Thread Management Alternatives

In a parallel environment, access to shared data structures must be serialized to ensure consistency and correctness. Our thread package uses spin locks for this purpose: when a processor tries to modify a data structure, it must first lock it to obtain exclusive access; if some other processor already holds the lock, the processor loops until the lock is released.

Locking implies dual concerns of latency and throughput [Kumar & Gonsalves 1977]. Latency is the cost of thread management under the best case assumption of no contention for locks. Throughput, on the other hand, is the rate at which threads can be created, started, and finished when there is contention. If part of thread management must be done serially, then no matter how many processors work on a problem, there will be some maximum rate of thread creation.

There are several ways of defining latency, with different implications for different types of applications. If an application keeps all of its processors continually busy, for instance by creating threads before they are needed, then any time spent in creating, starting, or finishing a thread is time that could have been spent doing other useful work. When a thread finishes, however, if there is no other work for the processor to do, the time spent deallocating the thread's data structures is unimportant. Instead, the relevant issues include how much a creating processor is delayed, since it has a thread to run, and how much time it takes for the created thread to begin running on a processor.

In the following subsections, we define five alternative thread management strategies, and describe some of the potential advantages and disadvantages of each approach. We then provide measurement and analytical comparisons of these alternatives.

3.1. Single lock: central data structures protected by one lock

The most obvious approach to thread management is to protect all data structures under a single lock. Once the lock is acquired by a processor, the processor is assured that it can modify any stored state. To perform a thread operation, a processor must first acquire the lock, then do what is needed to the shared data, and finally release the lock when done. In this way, only a single lock is needed per thread operation, but, since most of the thread management path is serialized, throughput is limited. In the typical scheme, idle processors loop checking the ready queue for work to do, causing useless contention for the ready queue lock; however, this can be avoided if idle processors check that the ready queue is not empty before acquiring the lock. (Ni and Wu [1985] present a different approach.)

3.2. Multiple locks: central data structures protected by separate locks

A somewhat more modular approach to locking is to separately protect each data structure with its own lock [Lampson & Redell 1980]. Each operation on the data structure can then be surrounded by a lock acquisition and release. For thread management, this involves separately locking each enqueue and dequeue operation on the ready queue, stack free list, and control block free list, the three shared data structures.

There is a basic tradeoff between latency and throughput in the choice between using a single lock or multiple locks in protecting shared data structures [Kumar & Gonsalves 1977]. Since less of the total thread activity is in a critical section, and since it is split among several locks, the maximum rate of thread creation is higher with multiple locks than with a single lock. There is a cost to this increased throughput, however: more lock accesses are needed, increasing latency.

3.3. Local freelist: per-processor free lists without locks

One way of avoiding locking is to maintain as much state as possible locally, with each processor. If each processor maintains its own free lists of control blocks and stacks, these need not be locked, since only one processor will access them. As before, there is a single shared ready queue whose accesses are locked.

The tradeoff between latency and throughput can be largely avoided by using local free lists. Since fewer lock acquisitions are needed per thread, latency is lower than with multiple locks, yet since only accesses to the ready queue are serialized, throughput is better.

Local free lists need to be balanced. Control blocks and stacks can migrate between free lists if the thread is created or started on one processor and finished on another. Thus, one free list can be empty, requiring the processor to obtain more space from the heap, while another free list has many entries. In the worst case, some processors only create and start threads (allocate structures), while other processors only finish them (deallocate structures). Without balancing the deallocated structures are never re-used; a separate stack and control block are needed for every thread. In contrast, with a centralized free list, only as many are needed as there are active (created or started, but not finished) threads.

It is inexpensive, however, to balance free lists by using a global pool and a threshold T on the maximum size of each list. When the size of a free list reaches the threshold, half the list can be returned to the global pool; when a free list empties, $T/2$ entries can be removed from the pool. The global pool must be locked, of course. For efficiency, it can be organized as a list of lists. The processing cost to balancing is thus one locked pool access amortized across at least $T/2$ free list accesses. Let P be the number of processors. An application using balanced local free lists will use no more than $O(P \times T)$ more space than one using a central list; the worst case occurs when one processor's free list is empty while all other free lists are almost full.

Thus, local free lists trade space for time. This tradeoff is practical for control blocks. Utilization of the pool lock is at most $O(PR/T)$, where R is the rate of thread creation on a single processor. To ensure that the pool lock is not a source of contention (which would inflate the overhead per free list access), we can set the threshold T to be equal to P. Control blocks are relatively small objects (in our implementation, roughly 100 bytes); provided P is not excessively large, using $100P$ bytes per processor is not onerous. If P is large, then a tree of pools could be used to limit the cost to balancing to $O(P/\log P)$ bytes per processor.

The tradeoff is not practical for stacks, however. Stacks are at least two orders of magnitude larger than control blocks. Even if sufficient memory were available, using that memory entails processing costs for initializing page tables and increased cache miss rates that could easily overwhelm the advantage gained from decreased locking. Instead, we use single element stack free lists. In this way, stacks need be allocated from the global pool only when a processor blocks a thread and then starts up a different thread, and deallocated only when a processor finishes a thread and then resumes another thread.

3.4. Idle queue: a central queue of idle processors

None of the algorithms described so far exploit parallelism in thread creation. The creating processor allocates and initializes the control block; when it is done, the starting processor allocates and initializes the stack. The cost of thread creation could be reduced if some of the work was done by idle processors in parallel with the creating processor.

In addition to a central queue of threads, we can maintain a central queue of idle processors. When there is a backlog of ready threads, there is no point to attempting parallel thread creation since all processors are already doing useful work. When a processor becomes idle and there is no backlog, it pre-allocates a control block and stack, puts itself on the idle queue, and spins on a local flag waiting for work. Thread creation then dequeues the idle processor, initializes the pre-allocated control block and stack, and sets that processor's flag, indicating that it now has a thread that is ready to run. Instead of processors searching for work, work searches for processors.

In fact, this approach does not alter the essentially sequential nature of thread creation. The idle processor must first queue itself before the creating processor can dequeue it, which in turn must set the flag before the idle processor can start running the thread. The critical path between the beginning of thread creation and when the thread starts running is reduced by doing some of the work (allocating structures, acquiring a lock, enqueueing) before the critical path begins. Since this adds complexity, and there is no benefit in the absence of idle processors, the effect is to trade off reduced latency when there are idle processors for increased latency when all processors are busy. Maximum throughput should be unchanged since two locked queue operations are still needed per thread life cycle. Wagner et al. [1988] describe a different way of using of idle processors to avoid work during blocking and resuming.

3.5. Local readyq: per-processor ready queues

Once free lists are made local, the ready or idle queue lock can become a serial bottleneck as the rate of thread creation or the number of processors increases [Dritz & Boyle 1987]. One way of increasing throughput is to divide the load on a single lock among several locks. An application of this idea is to keep a ready queue per processor. In this way, enqueueing and dequeueing threads can occur in parallel, with each processor using a different queue. There is again a tradeoff between latency and throughput in the choice between using one or more ready queues.

Unlike the case of control block free lists, unlocked local ready queues are inefficient even if balanced through a global pool. Runnable threads are a scarce resource. An idle processor might have an empty queue, yet a ready thread that the processor could run is in some other processor's queue, while the global pool is empty. Performance can be arbitrarily bad in any scheme where a processor can be idle indefinitely while there is even one ready thread in some other queue. In the worst case, P identical threads are created, but due to an imbalance, only $P - 1$ are started while one processor idles. The runtime would then be twice as long as with any of the centralized queueing strategies.

One simple way of avoiding indefinite idling is to lock each ready queue; each idle processor can then scan the ready queues for work, beginning with its own [Dritz & Boyle 1987]. If there is a ready thread, an idle processor will eventually find it. Processors can queue created threads locally, since balancing is achieved by idle processors. The worst case for this approach is when a single processor creates every thread, since that processor's queue would operate much as a central ready queue would, except that idle processors would have to waste time scanning for it. A simple way of avoiding this situation is for each processor to randomly choose a queue for each thread creation.

If each queue is equally likely to get a new ready thread, latency is bad when the number of runnable threads is near to the number of processors. There are two cases. Consider the cost of scheduling a thread onto a newly idle processor. If there are no ready threads, there is effectively no cost until a new thread is created. If there are ready but not running threads, any time spent finding a thread to run could have been spent running that thread. This time is small when there are many ready threads, because the idle processor will find the thread after scanning only a few queues; when there is only a single ready but not yet running thread the processor will have to examine on average half of the queues in order to find it. The cost of scheduling a newly created thread onto an idle processor is similar: the thread will be found quickly if there are many idle processors and more slowly if there are only a few.

One reason to have a one-to-one correspondence between processors and ready queues is to maintain locality. Presumably, migrating a newly active or resumed thread has a cost, due to increased cache misses. On the other hand, threads can only be maintained locally if there is a large backlog of ready threads [Eager et al. 1986]. While there are some message passing applications where this holds, there is little reason to create a new thread if it will simply run on the same processor that created it. In any case, the cost of migration is certainly application-specific.

If maintaining locality is unimportant, there is a tradeoff between latency and throughput in choosing the number of queues [Ni & Wu 1985]. Up to some point, throughput is higher with more queues, but the number of queues that must be scanned to find work, and thus the latency, is also higher. We set the number of queues equal to the number of processors for all measurements.

3.6. Measurement results

To validate our intuitions about the relative merits of the alternative approaches, we implemented each on a Sequent Symmetry Model A shared-memory multiprocessor. All code was written in C and compiled with Sequent's standard compiler, with the exception of the locking and context switching code, which was programmed in assembler. Our Symmetry has twenty Intel 386 processors, a shared bus, and a write-through cache coherency protocol [Lovett & Thakkar 1988]. The Symmetry has a timer with microsecond resolution that was used for all measurements. Table 3.1 contains times for sample Symmetry operations.

Operation	Runtime (μsec.)
Acquire and release a lock	5.6
Procedure call with no arguments	3.6
Each call argument	1
Iteration of null loop	2.5

Table 3.1: Runtimes for Symmetry operations (measured)

For all measurements, free lists were "warm started": sufficient control blocks and stacks were pre-allocated for use by the benchmark. Our purpose was to measure the relative merits of each alternative, rather than the efficiency of the underlying memory management. The cache was not warm-started, but we ran each benchmark long enough for this effect to become insignificant.

Figure 3.1 is the principal performance comparison: it shows the elapsed time in seconds for each thread management alternative to create, start, and finish one million "null" threads, for varying numbers of processors. Initially, P threads are created; each recursively creates a thread then finishes, allowing that processor to start up one of the waiting threads. The test terminates when each processor has executed $1M/P$ threads. For the multiple ready queue alternative, each newly created thread was added to a random queue to avoid biasing the results with the effect of locality. This test is not intended to be representative of a real parallel program, but it does expose the tradeoffs among the alternatives. (The one processor case shows the latency for a single thread in microseconds when there is no contention for locks.)

Figure 3.2 shows the inverse graph: the rate of thread creation (throughput) for each alternative, in units of 1000s per second.

Before examining the relative performance of the five alternatives, we note that each of them has quite good performance. Threads are only an order of magnitude more expensive than a procedure call, and 500 times less expensive than normal DYNIX process creation. Threads in PRESTO [Bershad et al. 1988a] cost 600 μsec. on the same Symmetry hardware, an order of magnitude worse than our threads although an order of magnitude better than DYNIX processes.

While PRESTO's speedup relative to DYNIX is due to using threads instead of processes, our speedup relative to PRESTO is due to attention to implementation details. We implemented PRESTO in C++; while this enhanced its ability to be modified [Bershad et al. 1988b], its C++ was first pre-processed into C, then compiled. This resulted in much less efficient code than could be achieved by direct coding in C. Another factor is that we stripped thread control blocks of all non-essential state, reducing the cost of initialization dramatically. We did not remove functionality: our thread package could be given PRESTO's user interface without sacrificing its performance.

Because our threads are inexpensive, the choice of alternatives has a large relative impact on both latency and throughput for applications with fine-grained parallelism. Specifically:

- Adding even a single lock acquisition into the thread management path can increase latency significantly. Locking each of the data structures separately results in a much higher latency than locking all data structures under the same lock. Using per-processor data structures to avoid locking is thus crucial to decreasing latency without sacrificing throughput.

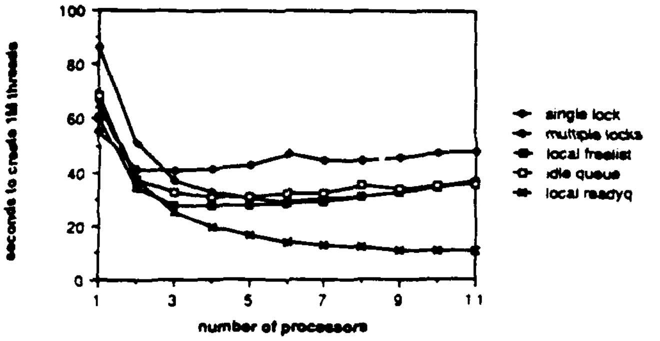


Figure 3.1: Principal results for thread management – elapsed time to create, start and finish 1M null threads (measured)

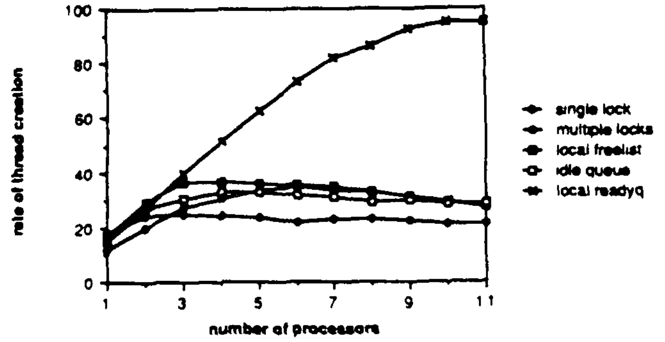


Figure 3.2: Rate of thread creation, 1000s of threads per second (inverse of Figure 3.1)

- Additional complexity results in a noticeable increase in latency. There are on the order of 100 instructions in the thread management path; adding even a few extra instructions impacts performance. For example, the idle queue strategy checks for idle processors on thread creation. If the idle queue is always empty, as in the measurements in Figure 3.1, it defaults to a normal ready queue. Even this simple a check markedly increases the cost of threads. This implies that thread management routines must be kept simple; enhancements that would otherwise seem plausible but add complexity are unlikely to work, since there is little computation to save, and it is easy to swamp the savings with increased overhead.
- A large portion of the thread management path is locked, since little work is required beyond manipulation of shared data. When all data is kept under a single lock, throughput is limited by contention for this lock. However, even with local free lists, the lock on the ready queue limits throughput to only a few concurrent thread operations. Only local ready queues can support high rates of thread creation.

When lock contention is not a problem, the bandwidth of the bus limits the thread creation rate. The throughput in Figure 3.2 levels out for the local ready queue alternative, even though there is no significant contention for locks. While the heavy bus demand per thread may be specific to the write-through cache protocol on the Symmetry, bus contention is likely to be a problem on any bus-structured shared-memory system.

In Figures 3.1 and 3.2, threads do no work except to create other threads. It is natural to ask whether the performance implications of the thread management alternatives would still be significant in the presence of user-mode computing. Figure 3.3 graphs thread creation rate as a function of the number of processors, when the amount of user work per thread averages 300 µsec, taken from a uniform distribution. This is representative of applications with fine-grained parallelism. Differences appear as the number of processors increases.

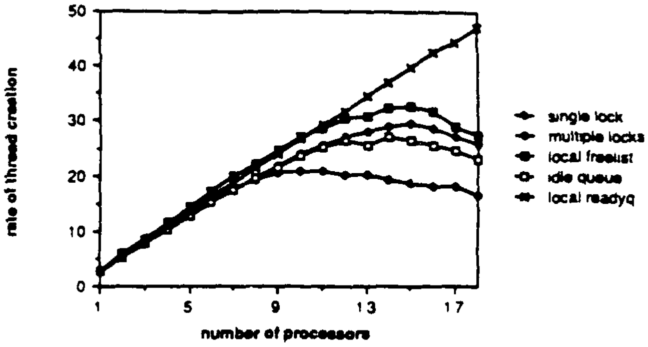


Figure 3.3: Rate of thread creation, 1000s of threads per second, user work = 300 µsec. (measured)

Figure 3.4 graphs thread cost in µsec. as a function of the number of runnable threads (parallelism). When there are fewer threads than processors, thread cost is taken to be the time to create and start running a new thread. The time to finish a thread is unimportant if the idling processor has no work to do. When there are as many or more runnable threads as processors, the cost is the sum of the time to create a thread plus the time to finish it and start a new thread. This difference in the definition of cost results in the jump in Figure 3.4 when the number of runnable threads reaches the number of processors. Note that the thread latency reported in Figure 3.1 with one processor corresponds closely to the latency reported in Figure 3.4 when there are more runnable threads than processors.

Thread cost was directly measured by taking timestamps before and after each thread was created and whenever a thread started or finished. Multiple creations were measured and averaged to improve accuracy. Creations and completions were synchronized to avoid measuring lock contention.

As expected, an idle queue is faster when there are idle processors, but slower when there are more runnable threads than processors. Thread creation is faster if an idle processor can be used to do work before the thread is created, but checking the idle queue incurs overhead even if it is not used. Whether a particular application will run faster with an idle queue depends on how much time it spends in each case.

The spike in the curve when using per-processor ready queues shows that finding a ready thread among many queues is expensive when the parallelism of the application is near to the number of processors, but the expense fades when more ready threads or more idle processors are available.

One area of further research is to examine hybrid thread management strategies to combine the advantages of some of the alternatives we have presented. For example, both central and per-processor ready queues could be used, by placing created threads in a local queue if the lock on the central queue is busy. As another example, a creating processor could probe randomly to find an idle processor, and if none were found, place the thread in a central queue. The drawback to any such approach is that complexity adds cost which may outweigh any benefits.

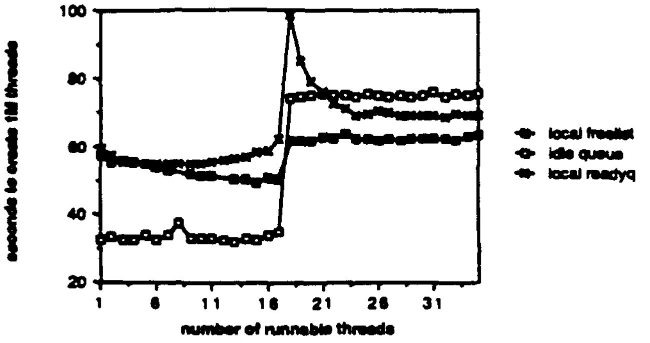


Figure 3.4: Latency (µsec.) vs. number of runnable threads, 18 processors (measured)

3.7. Analytical explanation of Figure 3.4

We now derive a formula that explains in detail the spike for the per-processor ready queue alternative in Figure 3.4. When there are idle processors, we need to know the time between the queueing of a ready thread and the dequeueing of that thread by an idle processor; when there is a backlog of ready threads, we need to know how long it takes a newly idle processor to find one.

Let $E(r,q)$ be the expected number of queues examined by a newly idle processor to find one of r ready threads, which are randomly distributed among q queues. Without loss of generality, let the queues be numbered from 1 to q, let threads be numbered from 1 to r, let i_j be the queue containing the jth thread, and let the idle processor begin searching with queue 1. The idle processor must examine the number of queues equal to the lowest numbered non-empty queue. The number of ways of putting r threads into q queues is q^r.

$$E(r,q) = \frac{1}{q^r} \sum_{i_1, i_2, \cdots i_r = 1}^{q} \textit{minimum of } (i_1, i_2, \cdots i_r)$$

We can separately sum when each i_j is the minimum. When more than one thread is at the minimum, we count the value once in the sum for the least numbered thread. Thus, the value of i_j is counted only if for all $k<j$, $i_k>i_j$, and for all $k>j$, $i_k \geq i_j$.

$$E(r,q) = \frac{1}{q^r} \left[q + \sum_{j=1}^{r} \sum_{i=1}^{q-1} i(q-i+1)^{r-j}(q-i)^{j-1} \right] \quad (3.1)$$

By symmetry, Equation 3.1 also holds when there are more processors than runnable threads. Let r be the number of idle processors, let i_j be the queue currently scanned by the jth idle processor, and let the newly created thread be put into queue 1. Then the processor that actually dequeues the thread will have to look through $E(r,q)$ queues, after the thread is queued, to find it.

Figure 3.5 graphs Equation 3.1 for 18 processors. To compare to Figure 3.4, the x-axis is the number of runnable threads, rather than the number of ready but not running threads or the number of idle processors. Noting that part of the spike in Figure 3.4 is due to the difference in the measurements when there are idle processors or not, Figures 3.4 and 3.5 correspond well.

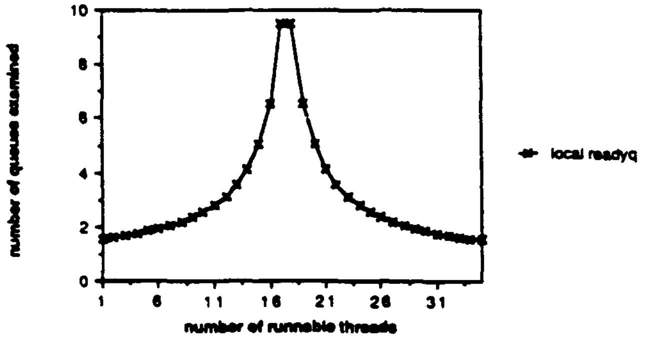


Figure 3.5: Queues examined vs. number of runnable threads, 18 processors (Equation 3.1)

The above analysis assumes that events occur one at a time. Since finding a ready thread among a number of queues can take a non-trivial amount of time, it is reasonable to consider what happens when another thread is created or another processor becomes idle during the interim. Suppose another thread is created before an idle processor finds one of the r ready threads. Let C be the cost of finding a thread in this situation. If the new thread is the one that is found, then C is no better than if the new thread had been there all along. If a different thread is found, then C is no worse than if the new thread is ignored. In other words, $E(r+1,q) \leq C \leq E(r,q)$. Similarly, if another processor becomes idle in the interim, provided $r \geq 2$, the combined cost for both processors to find threads is $E(r,q) + E(r-1,q)$, assuming the processors do not contend for the same queue, independent of which processor finds a ready thread first.

4. Spinlock Management Alternatives

If a processor finds a thread management lock busy, it must spin wait for the lock to be released. Since any other work the processor might do instead is also controlled by a lock, the processor does not have the option of doing other work while it is waiting.

At the user level, a thread does have a choice between spinning for a busy lock or blocking, relinquishing the processor to do useful work while the lock is busy. Since finding that work requires access to thread management data structures, however, blocking at the user level may result in spinning in a thread routine.

Spin-waiting has a hidden cost. Processors doing useful work may be slowed by processors that are merely waiting for a lock, due to bus contention. As a result, adding to the number of processors executing an application may in fact slow it down by increasing the average number of spinning processors. Worse, the more spinning processors, the more the processor holding the lock is slowed, increasing the effective size of the critical section, resulting in even more waiting processors.

Here we evaluate three different approaches to spin-waiting.

4.1. Hardware description

On the Symmetry Model A, each processor has its own cache; provided all of its memory references can be satisfied out of that cache, a processor's progress is independent of the activity of other processors. Whenever a processor reads data that is not in its cache, it must wait for the data to come from memory via the bus; with a write-through protocol, a processor may also have to wait for writes to be sent to memory. In both cases, the processor's progress can be slowed by bus contention.

The Symmetry has a basic test-and-set instruction, xchgb (exchange byte), that atomically reads a memory location and writes in a new value. The atomicity of the xchgb operation is enforced by the bus: a copy of the memory location is brought into the processor's cache, modified there, and then written back to memory. Any requests for that memory location in the interim are delayed until the processor is done modifying it [Lovett & Thakkar 1988].

The Sequent locking protocol is as follows: To lock, a processor exchanges in a 1. If the old value was a 0, it got the lock; if the value was a 1, the lock was already held by someone else, and the processor must try again. In either case, the value is 1 afterwards. The lock is released by exchanging in a 0; this allows some other processor to get a 0 back in exchange for a 1. There are several potential protocols for spin-waiting, which are described below.

4.2. Spin on xchgb

The simplest way to implement spin-waiting is for each processor to loop on the xchgb instruction until it succeeds. The drawback to this approach is that every xchgb instruction consumes bus resources, whether or not it succeeds [Sequent 1988]. A copy of the lock must be brought into the processor's cache; since the lock is written whether or not it is acquired, any copy of the lock in another cache is invalidated. As additional processors spin on the lock, the holder of the lock is slowed both because the bus is busier and because to free the lock it must contend with atomic operations of processors uselessly trying to acquire the lock.

4.3. Spin on memory read

An alternative would be for each processor to try to acquire the lock once; if this fails, the processor can spin reading the lock memory location. As long as the value is 1, the lock is still held. Looping on a read is done in the cache, avoiding bus traffic. When the lock is released, the cache copy will be invalidated; the spinning processor will see the value change to 0, and can then try to acquire the lock using an xchgb operation. Sequent's runtime library uses this implementation [Sequent 1988].

A problem arises when there are a number of processors waiting for a small critical section. When the lock is freed, every spinning processor's copy is invalidated, causing each processor to miss in turn. The first to try to acquire the lock succeeds. Any processor that reads the value before this occurs will see a 0 and will attempt to acquire the lock (and fail); any processor that reads the value afterwards will see a 1 and will return to looping in its cache. Unfortunately, each processor that does an unsuccessful xchgb operation invalidates all cache copies, forcing all processors that had seen a 1 to read miss again. After each such operation, virtually every spinning processor must contend for the bus, some still waiting to do an xchgb and some waiting for a read miss. Eventually, the last processor to have seen a 0 will attempt to acquire the lock and fail; each spinning processor can then read miss and quiesce, looping in its cache.

The performance of this algorithm, therefore, improves as the critical section gets longer, assuming that contention does not increase. After the lock is released and before quiescence, each spinning processor spends most of its time with a pending bus request; any normal bus request during this time will be correspondingly delayed. After quiescence, the spinning processors place no load on the bus, allowing the processor holding the lock to progress unhindered. With longer critical sections, the initial degradation is less significant. By contrast, spinning on the xchgb instruction degrades bus performance evenly throughout the critical section.

4.4. Ethernet-style backoff

The source of the difficulty is that there is a cost to attempting to acquire the lock. A generic solution to problems of this sort is to have each processor estimate its likelihood of success, and only try the lock when the probability is high. The estimate can be made from experience. The more times a processor has tried and failed, the more likely it is that many processors are spinning for the lock. When the lock is released, then, instead of every processor rushing to try to get it, each waits a period of time dependent on the number of past failures. If the lock is still free after this period, then the probability of success is high enough to try the lock. We used this algorithm for our measurements in Section 3.

The analogy with Ethernet is revealing. In the Ethernet protocol, a processor can start a network transmission in any time slot that the network is free [Metcalfe & Boggs 1976]. If two try to start transmitting in the same slot, both fail and must be retried later. To avoid further collisions, the length of time before retrying depends on the number of collisions encountered so far. In our case, when a number of processors simultaneously try to acquire a lock, one will succeed, but its progress will be slower than if there were no collisions.

The downside to Ethernet-style protocols is that they are unfair. A processor that has just arrived is more likely to acquire the lock (or network) than one who has been waiting, and failing, for some time. Spinning on a test-and-set instruction and spinning on a copy of the lock location are both probabilistically fair; each spinning processor has an equal likelihood of getting the lock, even though the possibility of indefinite starvation exists. Lock fairness is sometimes important to an application.

Another drawback of the backoff algorithm is that it takes longer for a spinning processor to acquire a newly free lock. The processor must check the lock value, delay, and check it again before trying the lock. Once the lock is acquired, however, the processor will proceed faster, relatively unimpaired by other spinning processors.

Even using this algorithm, there will be processor degradation when there are large numbers of spinning processors. When the lock is released, every spinning processor encounters a cache miss. After this initial miss, most processors delay locally until some other processor has acquired the lock, and then miss again to see that the lock has been acquired. With enough spinning processors, the bus can be saturated with these misses, slowing down the processor executing in the critical section.

These cache misses can be avoided. A processor can delay whenever it reads the lock value as busy. If the lock is not busy, the processor can immediately try to acquire it. Thus, spinning processors miss their cache every time the delay period expires, rather than every time the lock is released. This is analogous to the Ethernet notion of persistence [Metcalfe & Boggs 1976]. A result of this variation is an even greater delay between when a lock is released and when a spinning processor will acquire the lock. Nevertheless, this type of spin-waiting may be appropriate for systems without hardware-coherent private caches. In this case, spinning on a memory read until the lock is released is impractical since each read consumes bus resources; backoff adapts the frequency of reads to the number of waiting processors.

While most practical applications will not waste large numbers of processors, this can be a problem with idle processors polling a central or distributed ready queue. When a ready thread is queued, if each idle processor rushes to acquire the lock, bus saturation will result. Even if each idle processor delays after observing that a thread is queued, then makes sure that it is still queued, each idle processor will still perform a cache miss, hurting performance for large numbers of idle processors.

If idle processors are kept on a queue, this problem does not occur. Each idle processor spins on a local flag. When a thread is created, only one processor's flag is modified; every other processor continues spinning without even a cache miss. The performance advantage of having work look for processors instead of processors looking for work will therefore be more important in systems with large numbers of processors. This effect can be seen in Figure 3.4; the cost of the central ready queue is higher when there are only a few runnable threads, since there are more idle processors spin-waiting for work to appear in the ready queue.

4.5. Measurement results

Figure 4.1 shows the elapsed time to increment and test a shared counter in a critical section 1 million times, for each method of spin-waiting. Each processor executed a loop: wait for the lock, increment the counter, and release the lock. If spin-waiting did not slow the processor holding the lock, the elapsed time for twenty processors would be no more than for one.

The magnitude of this effect is striking. Both spinning on the xchgb instruction and spinning on the copy of the lock degrade processor performance badly for even a moderate number of spinning processors. For small critical sections, in either alternative, every spinning processor spends all of its time doing cache read misses or atomic xchgb operations, consuming bus resources as fast as possible. By contrast, the backoff algorithm results in only slight degradation for less than ten spin-waiting processors.

Figure 4.2 shows the effect of increasing the size of the critical section for each algorithm. In addition to incrementing a counter, the critical section contained varying amounts of other work. We then normalized the time for the counter to be cooperatively incremented by eight processors by the time for one processor. This measures relative processor speed. Again, if spin-waiting did not slow the processor holding the lock, one processor would not be faster than eight, and the relative processor speed would always be equal to 1. As expected, spinning on memory read degrades performance less as the size of the critical section grows, while spinning on the xchgb instruction degrades performance evenly throughout the critical section.

To test the tradeoff between processor degradation and the delay in acquiring a newly released lock, we measured the elapsed time for a number of processors to each increment a shared counter within a critical section. Once a processor acquired the lock and bumped the counter once, it was set to loop until all processors were done. This test is indicative of the cost of using a lock for barrier synchronization. Figure 4.3 shows the elapsized time divided by the number of processors. If there is no processor degradation or delay in acquiring the lock, the elapsed time to achieve the barrier should increase linearly with each additional processor; the normalized curve in Figure 4.3 should be flat.

Figure 4.3 shows that for small numbers of processors, spinning on the xchgb instruction is fastest, since a processor immediately acquires the lock when it is released. As more processors are added, however, this benefit is outweighed by the degradation of the processor holding the lock. The backoff algorithm shows a similar curve to spinning on a memory read, but for a different reason. Initially, many processors are queued for the lock; this leads spinning processors to guess large delay times. As more processors acquire the lock, there are fewer queued processors, and the delays become inappropriate.

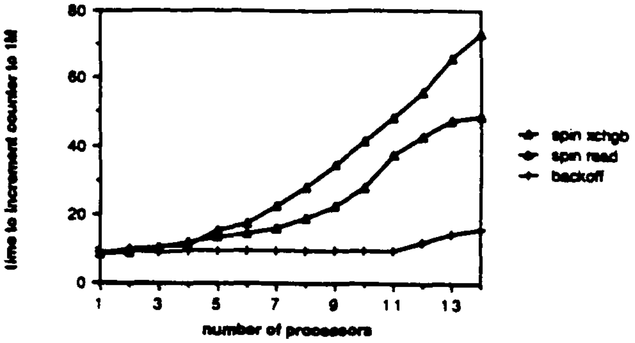


Figure 4.1: Principal results for spin-waiting: elapsed time to increment a shared counter to 1,000,000 (measured)

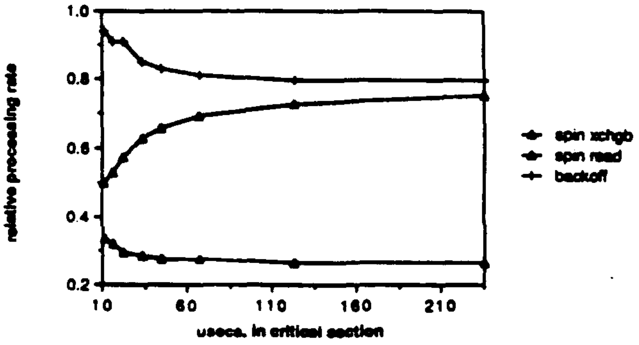


Figure 4.2: Relative processor speed (8 processors to 1 processor) vs. critical section size (measured)

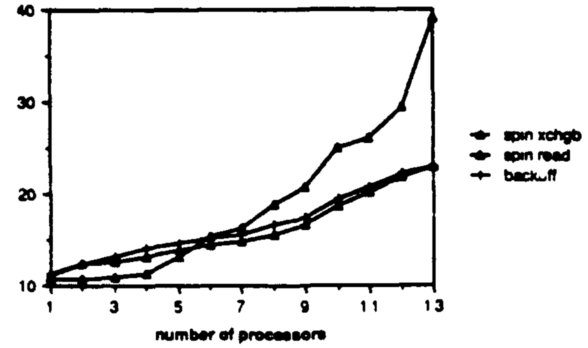


Figure 4.3: Normalized time (μsec. per processor) to achieve barrier (measured)

Processors doing work are slowed proportional to the number of times they access the bus. Thus, the results of these tests depend somewhat on the content of the critical section. However, since the purpose of a critical section is to serialize modifications to shared data, its code is likely to be bus intensive. Our measurements indicate that almost half of the bus service demand of thread management is due to the critical section. Further, thread management critical sections also tend to be small. For example, enqueueing or dequeueing a ready thread in a critical section both take less than 10 µsec., roughly the same as for Figure 4.1.

4.6. Implications for other systems

The Symmetry Model A has a write-through protocol: when a processor modifies a location, the value is written to memory and all old copies of the location in other caches are invalidated. There is a cost to spin-waiting, even in architectures with a write-back cache coherency protocol. In a write-back protocol, the value is stored in the cache and later written to memory when the cache block is replaced. There are two major approaches to keeping other caches consistent with the new value: all old copies in other caches can either be invalidated or updated with the new value (distributed-write) [Archibald & Baer 1986].

In the case of an invalidation-based write-back protocol, the spin-waiting alternatives have much the same effect as with write-through. If processors spin on the atomic test-and-set operation, the valid copy of the lock bounces from cache to cache, consuming bus resources. Provided more than one processor is spin-waiting, when one processor tries the lock, it invalidates every other cache copy, requiring the lock value to be copied to the cache of the next processor to try the lock. Spinning on a memory read does not solve this problem, since the cache copy of a looping processor is still invalidated, resulting in a cache miss, by each successive processor trying to acquire the lock. The Sequent Symmetry Model B, the successor to the architecture we used for our measurements, uses such a protocol.

The performance with distributed write-back is better, but it does not eliminate the problem. When a processor performs an atomic operation, every cache with an old copy is updated with the new value. If processors spin on the atomic operation, the bus can be saturated doing these updates. If processors spin on the memory read, however, each cache is kept up-to-date, eliminating the cascade of cache misses as each spinning processor tries to acquire the lock. The rush of processors to try the lock when it is first released still results in some bus traffic for distributing each update, but quiescence will occur faster. Since the backoff algorithm reduces the number of lock attempts, it reduces the bus load due to spinning even further.

A hardware mechanism for queueing processors without consuming bus resources would also solve this problem. In fact, the Symmetry has such a mechanism, but it is less than completely useful. While one processor is performing an atomic operation, any other processor attempting to access that memory location is delayed before using the bus [Sequent 1988]. Unfortunately, only single instructions can be made atomic; it is rare in practice to be able to complete a critical section in one instruction.

5. Analytical Results

We developed a queueing network model of our thread package to demonstrate that the combination of processor degradation due to bus contention and the effect of lock contention can account for our measurements. We used the validated model to project the performance of our package under varying conditions.

Our model is hierarchical. The low level model represents the effect of bus contention on processor speed. The high level model represents the effect of lock contention on throughput and response time. Since processor speed affects the amount of lock contention and the number of spinning processors affects bus contention and thus processor speed, we iterate between levels to convergence. We describe the two sub-models in more detail below.

5.1. Modelling bus contention

In the low level model, we represent each processor as a customer in a closed queueing network. The network has two service centers: a queueing center for the bus and a delay center for non-bus activity. Each processor spends some of its time referencing memory through the bus and thus contending with other processors also using the bus, and some of its time processing out of its cache, independent of the activity of other processors. Processor speed is degraded by the percentage of time spent queueing, but not in service, at the bus.

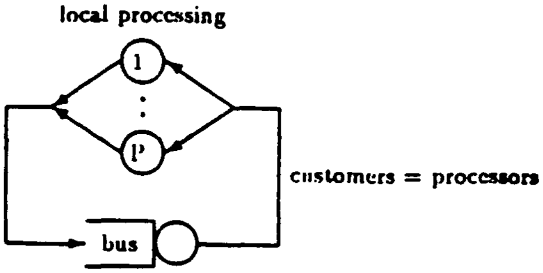


Diagram 5.1: Low level model of bus contention

This model is an approximation of the real bus mechanism, which is considerably more complex [Lovett & Thakkar 1988]. At moderate loads, our model will be pessimistic by predicting more contention than is actually experienced. Because of the regularity of the time each processor spends computing between accesses to the bus, if two processors collide at the bus, they are unlikely to collide at their next visit. Our model assumes that arrivals are more nearly independent.

There are three components to bus utilization. A processor can be executing user code, thread management code, or spin-waiting, each with different service demands on the bus. Given these service demands and the ratio of time each processor spends in each type of activity, we determine the aggregate service demands at the bus and at the delay center and use these aggregate demands to solve the model.

Since it is difficult to analytically determine the bus demand of a section of code, we determine a portion of it inductively from measurements. We provide each processor with its own copy of all data structures; we then run the code in parallel on each processor. Since there is no shared data, there can be no contention for software resources; any delay experienced by a processor relative to when it is running the code by itself must be due to contention for hardware resources, such as for memory or the bus. We then match a curve from our model of the bus to the measured curve and use the result as the service demand for that section of code. The curves matched well in practice, deviating only at moderate loads, as expected.

Since bus contention may disproportionately impact the critical section execution time, affecting lock contention in the high level model, we used this approach separately for the critical section and non-critical section code within thread management. The critical section code turns out to account for much of the bus demand of thread management.

Even though it could affect bus usage, we did not include in our model the effect of different numbers of processors on cache hit ratios. When a processor writes a location, the Symmetry updates both memory and that processor's cache. As a result, on a single processor, data that is both written and read will tend to stay in the cache, avoiding cache misses. When multiple processors read and write shared data, the cache copies of the data will be repeatedly invalidated as different processors update it, resulting in more cache misses than in the single processor case. Our model therefore underestimates bus demand, making it optimistic, especially as the bus nears saturation.

The bus demand of spinning processors was also determined inductively. *P* processors were set to run the critical section with separate copies of the data structures; by the experiment described above, we know the bus service demand of these processors. *Q* processors were set to run a shared copy of the critical section; one of these processors has the normal bus service demand, and $Q-1$ spin-wait. By measuring the processor degradation of the *P* copies, we can determine the aggregate bus demand of the $Q-1$ spinning processors. A two class model was used, one class representing processors executing critical sections and one representing spinning processors. Only the response time of the processors executing the critical section is important.

The bus demand, at least for the backoff algorithm, is linear with the number of processors. While there is no *a priori* reason for this, it intuitively makes sense. The effect of adding a spinning processor with the backoff algorithm is to add two cache misses per execution of the critical section. The bus demand of other processors is relatively unaffected. While this invariance would also hold for the spin on xchgb algorithm, it is less true when processors spin on memory reads, because the cascade of cache misses is longer for every processor when more processors are spinning. Note that the graphs in Section 4 could be used to infer the bus demand of spinning processors. We did not choose this approach because there is a correlation between when the processor holding the lock and when the processors spinning on the lock use the bus. The curve for the backoff algorithm in Figure 4.1, e.g., is similar to that of an optimistic asymptotic bound.

5.2. Modelling lock contention

In the high level model, we represent each lock in the thread management path by a separate queueing center. Processing time spent not holding a lock is modelled as a delay center. Service demands were directly measured, then the part of each service demand due to bus accesses was inflated by the bus response time of the low level model. As in the low level model, each processor is represented as a single customer in a closed class. By solving this model, we can determine the average amount of time each processor spends spin-waiting for a lock versus executing thread operations or user code. This ratio is then used as an input to the low level model. (Note that it is a simple matter to add queueing centers if the application-level code does further locking.)

If the time between thread operations is deterministic, our model is pessimistic at moderate loads. As for the bus, if two processors collide at a lock, the effect of deterministic processing times is to reduce the likelihood that they will collide at the next visit. Figure 3.2 shows this effect. The curves are similar in shape to asymptotic optimistic bounds, since the processing time to do each thread operation is deterministic. Figure 3.3 does not show this effect, since the user computation for each thread was randomly chosen from a uniform distribution.

Our model does not explicitly represent an application's distribution of parallelism, although Figure 3.4 shows that this affects performance. We chose not to include this in our model since the distribution and the effect of lock queueing delay on that distribution are almost always application-dependent.

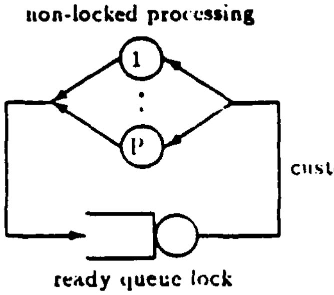


Diagram 5.2: High level model of lock contention for the local freelist alternative

Given the distribution, the model could be evaluated separately for each population of threads; these separate evaluations could then be averaged, weighted by the proportion of time for that population. The population of the high level model should be the minimum between the number of processors and the number of threads, reflecting the number of active processors. The population of the low level model should be set similarly, except that since idle processors consume bus resources, a second class should be added to represent them.

This method of separate evaluations ignores the fact that lock contention can only occur when the parallelism is being incremented or decremented; we believe that any distortion introduced by the adaptive nature of the mechanism will be outweighed by the effects of lock and bus contention. Ni and Wu [1985] also discuss this issue.

5.3. Comparison with measured results, and projections

Figure 5.1 compares our model results with our measurement results previously reported in Figure 3.3. We modelled two alternatives: per-processor ready queues (local readyq) and per-processor free lists with a central ready queue (local freelist). Our model agrees well with the measurements, within 5% except for the central ready queue with 18 processors. The model predicts the shape of the curve, but is somewhat optimistic; this appears to be due to underestimating the bus demand, which is important in determining the effective size of the critical section. The model does capture the difference between the alternatives.

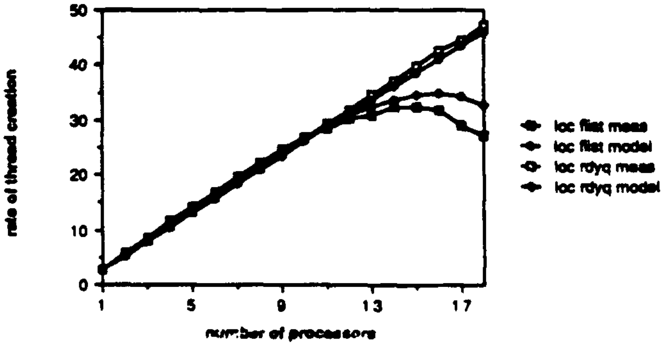


Figure 5.1: Comparison of analytic and measured results from Figure 3.3

Having validated our model, we used it to investigate the effect of varying key parameters. Figure 5.2 shows throughput with 20 processors as a function of the amount of user computation per thread. As we would expect, as an application uses finer-grained parallelism (smaller amounts of computation per thread), the central lock on the ready queue becomes a bottleneck. For sufficiently coarse-grained parallelism, the performance of the thread package ceases to matter. In the limit, even DYNIX processes could be used.

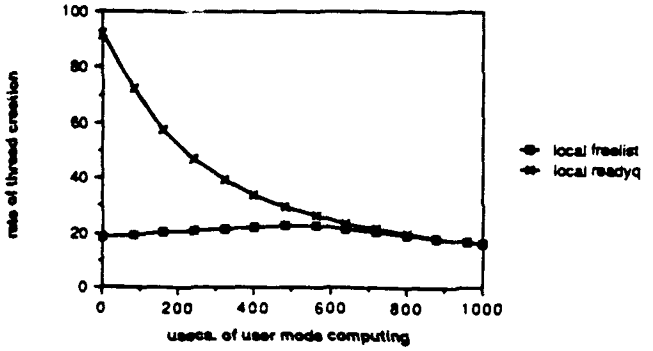


Figure 5.2: Thread creation rate vs. μsec. of user computation per thread, 20 processors, bus load = 5% (analytic)

Contention for the bus can also reduce the difference between the alternatives. Figure 5.3 shows throughput as a function of the percentage usage of the bus by each thread. As the bus usage increases, the bus limits the throughput with local ready queues, but it also limits the throughput with the central ready queue, since bus contention inflates the critical section time.

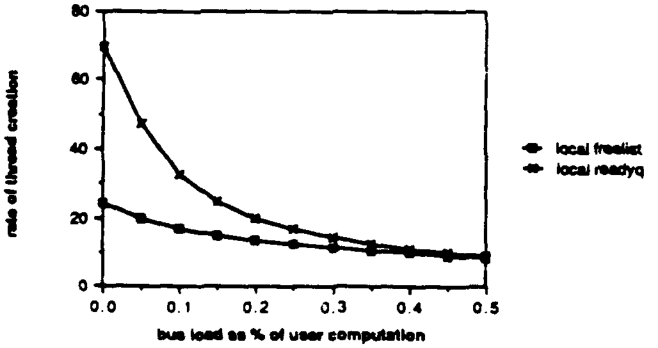


Figure 5.3: Thread creation rate vs. bus load, user work = 200 μsec., 20 processors (analytic)

On the other hand, the central ready queue lock can again limit throughput even for more coarsely-grained parallelism, given a sufficient number of processors. Figure 5.4 shows the throughput as a function of the number of processors when threads each compute for 2 milliseconds. The sharp dropoff for the central ready queue alternative shows the inherent instability of a system where spinning processors consume resources.

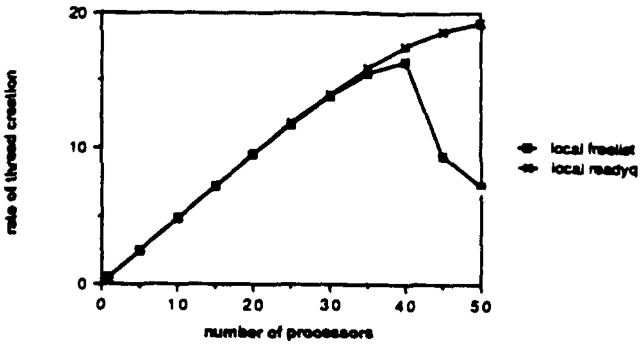


Figure 5.4: Thread creation rate vs. number of processors, user work = 2 msec. (analytic)

6. Conclusions

Threads have become a common element of new languages and operating systems. Efficient thread management is critical to achieving good performance from parallel applications. We have studied the performance implications of several thread management and locking alternatives. We showed that:

- It is possible to implement a fast thread package. Simplicity is crucial for this.
- For fine-grained parallelism, small changes in data structures and locking have a large effect on both latency and throughput.
- Per-processor data structures can be used to improve throughput; if a resource is not scarce, localizing data can avoid locking, improving latency as well.
- Spin-waiting can delay not only the processor waiting for a lock, but other processors doing work. This appears to be independent of the cache coherency protocol.
- An Ethernet-style backoff algorithm can reduce the cost of spin-waiting.
- A simple queueing model can accurately predict the effect of a combination of factors on the performance of shared-memory multiprocessors.

An area of future research is to determine the extent to which our results, developed in the context of thread management systems, also apply to application programs that exploit fine-grained parallelism on shared-memory multiprocessors.

Acknowledgements

We would like to thank Dave Wagner for suggesting that an Ethernet-style algorithm might solve the spin-waiting problem.

References

[Accetta et al. 1986]
M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. *Proc. Summer 1986 USENIX Technical Conference and Exhibition*, June 1986, pp. 93-112.

[Archibald & Baer 1986]
J. Archibald and J.L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, vol. 4, no. 4, Nov. 1986.

[Bach & Buroff 1984]
M.J. Bach and S.J. Buroff. Multiprocessor UNIX Operating Systems. *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, Oct. 1984, pp. 1733-1749.

[Bershad et al. 1988a]
Brian Bershad, Edward Lazowska, and Henry Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice and Experience*, vol. 18, no. 8, Aug. 1988, pp. 713-732.

[Bershad et al. 1988b]
Brian Bershad, Edward Lazowska, Henry Levy, and David Wagner. An Open Environment for Building Parallel Programming Systems. *Proc. ACM/SIGPLAN PPEALS 1988*, pp. 1-9.

[Dritz & Boyle 1987]
Kenneth W. Dritz and James M. Boyle. Beyond "Speedup": Performance Analysis of Parallel Programs. Technical Report ANL-87-7, Mathematics and Computer Science Division, Argonne National Laboratory, Feb. 1987.

[Eager et al. 1986]
Derek Eager, Edward Lazowska, and John Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, vol. 12, no. 5, May 1986, pp. 662-675.

[Edler et al. 1988]
Jan Edler, Jim Lipkis, and Edith Schonberg. Process Management for Highly Parallel UNIX Systems. Ultracomputer Note #136, April 1988.

[Hoare 1978]
C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, vol. 21, no. 8, Aug. 1978, pp. 666-677.

[Holt 1982]
R. Holt. A Short Introduction to Concurrent Euclid. *SIGPLAN Notices*, vol. 17, May 1982, pp. 60-79.

[Jul et al. 1988]
Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, vol. 6, no. 1, Feb. 1988, pp. 109-133.

[Kumar & Gonsalves 1977]
B. Kumar and Timothy Gonsalves. Modelling and Analysis of Distributed Software Systems. *Proc. 7th ACM Symposium on Operating Systems Principles*, Dec. 1977, pp. 2-8.

[Lampson & Redell 1980]
B.W. Lampson and D.D. Redell. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, vol. 23, no. 2, Feb. 1980, pp. 104-117.

[Lazowska et al. 1984]
Edward Lazowska, John Zahorjan, G. Scott Graham, and Kenneth Sevcik. Quantitative System Performance. Prentice-Hall, 1984.

[Lovett & Thakkar 1988]
Tom Lovett and Shreekant Thakkar. The Symmetry Multiprocessor System. *Proc. 1988 International Conference on Parallel Processing*, pp. 303-310.

[Metcalfe & Boggs 1976]
Robert Metcalfe and David Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, vol. 19, no. 7, July 1976, pp. 395-404.

[Mundie & Fisher 1985]
D.A. Mundie and D.A. Fisher. Parallel Processing in Ada. *IEEE Computer*, Aug. 1985, pp. 20-25.

[Ni & Wu 1985]
Lionel Ni and Ching-Fern Wu. Design Trade-offs for Process Scheduling in Tightly Coupled Multiprocessor Systems. *Proc. 1985 International Conference on Parallel Processing*, pp. 63-70.

[Scott et al. 1988]
Michael Scott, Thomas LeBlanc, and Brian Marsh. Design Rationale for Psyche, a General Purpose Multiprocessor Operating System. *Proc. 1988 International Conference on Parallel Processing*, August, 1988.

[Sequent 1988]
Sequent Computer Systems, Inc. Symmetry Technical Summary.

[Thacker et al. 1988]
Charles Thacker, Lawrence Stewart, and Edward Satterthwaite Jr. Firefly: A Multiprocessor Workstation. IEEE Transactions on Computers, vol. 37, no. 8, Aug. 1988, pp. 909-920.

[Vandevoorde & Roberts 1988]
Mark Vandevoorde and Eric Roberts. WorkCrews: An Abstraction for Controlling Parallelism. Digital Equipment Corporation Systems Research Center, 1988.

[Wagner et al. 1989]
David Wagner, Edward Lazowska, and Brian Bershad. Techniques for Efficient Shared-Memory Parallel Simulation. *Proc. Performance '89 / ACM SIGMETRICS 1989*.

[Zahorjan et al. 1988]
John Zahorjan, Edward Lazowska, and Derek Eager. Spinning Versus Blocking in Parallel Systems with Uncertainty. *Proc. International Seminar on the Performance of Distributed and Parallel Systems*, North Holland, Dec. 1988.

The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors

Thomas E. Anderson

Department of Computer Science
University of Washington
Seattle, WA 98195

August 1989

Abstract

Most shared-memory multiprocessor architectures provide hardware support for making mutually exclusive accesses to shared data structures. This support usually consists of instructions that atomically read and then write a single memory location. These atomic instructions are used to manipulate *locks*; when a processor is accessing a data structure, its lock is busy, and other processors needing access must wait.

For small critical sections, spinning (or "busy-waiting") for a lock to be released is more efficient than relinquishing the processor to do other work. Unfortunately, spin-waiting can slow other processors by consuming communication bandwidth.

This paper examines the question: Are there efficient algorithms for software spin-waiting given hardware support for atomic instructions, or are more complex kinds of hardware support needed for performance?

We consider the performance of a number of software spin-waiting algorithms. Arbitration for control of a lock is in many ways similar to arbitration for control of a network connecting a distributed system. We apply several of the static and dynamic arbitration methods originally developed for networks to spin locks.

We also propose a novel method for explicitly queueing spinning processors in software by assigning each a unique sequence number when it arrives at the lock. Control of the lock can then be passed to the next processor in line with minimal effect on other processors.

Finally, we examine the performance of several hardware solutions that reduce the cost of spin-waiting.

Index Terms – multiprocessor, architecture, locking, performance, cache coherence

1. Introduction

Many shared-memory multiprocessors have been designed in the past few years. The Sequent Symmetry [Lovett & Thakkar 1988], Alliant FX [Perron & Mundie 1986], and the BBN Butterfly [BBN 1985] are among the more commercially successful; research vehicles include the DEC SRC Firefly [Thacker et al. 1988], Illinois Cedar [Gajski et al. 1983], IBM RP3 [Pfister et al. 1985], and the Wisconsin

---

This material is based on work supported by the National Science Foundation (Grants No. CCR-8619663, CCR-8703049, and CCR-8700106), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

Author's address: Department of Computer Science FR-35, University of Washington, Seattle WA 98195; (206) 543-2675; *tom@cs.washington.edu*.

Multicube [Goodman & Woest 1988].

In shared-memory multiprocessors, each processor can directly address memory that can also be addressed by all other processors. This uniform access requires some method for ensuring mutual exclusion: the logically atomic execution of operations (critical sections) on a shared data structure. Consistency of the data structure is guaranteed by serializing the operations done on it.

Since pure software mutual exclusion is expensive [Lamport 1987], virtually all shared-memory multiprocessors provide some form of hardware support for making mutually exclusive accesses to shared data structures. This support usually consists of instructions that atomically read and then write a single memory location. All of the multiprocessors mentioned above support atomic instructions, although some, most notably the Multicube, also provide other mechanisms [Goodman et al. 1989].

Atomic instructions serve two purposes. First, if the operations on the shared data are simple enough, they can be encapsulated into single atomic instructions. (Herlihy [1988] discusses the computational power of atomic instructions for building parallel algorithms.) Mutual exclusion is directly guaranteed in hardware. If a number of processors simultaneously attempt to update the same location, each waits its turn without returning control back to software.

A *lock* is needed for critical sections that take more than one instruction. Atomic instructions are used to arbitrate between simultaneous attempts to acquire the lock, but if the lock is busy, waiting is done in software. When a lock is busy, the waiting process can either block, relinquishing the processor to do other work, or spin ("busy-wait") until the lock is released. Even though spin-waiting wastes processor cycles, it is useful in two situations: if the critical section is small, so that the expected wait is less than the cost of blocking and resuming the process, or if no other work is available.

This paper examines the question: are there efficient algorithms for software spin-waiting for busy locks given hardware support for atomic instructions, or are more complex kinds of hardware support needed for performance? (Jayasimha [1987] and Agarwal and Cherian [1989] have looked at the related issue of efficient spin-waiting for data dependencies.)

We show that the simple approaches to spin-waiting for busy locks have poor performance [Anderson et al. 1989]. Spinning processors can slow processors doing useful work, including the one holding the lock, by consuming communication bandwidth. This performance penalty occurs if processors spin by continuously trying to acquire the lock; it also occurs for small critical sections if processors spin reading the (cached) lock value and try to acquire the lock only when it is released.

We consider the performance of several software spin-waiting alternatives. Although the analogy is not perfect, arbitration for control of a lock is in many ways similar to arbitration for permission to transmit on carrier-sense multiple-access (CSMA) networks. In both there is a cost when either zero or more than one waiting processor attempts to acquire the resource. A number of arbitration mechanisms have been proposed for CSMA networks, including statically assigned slots (BRAM [Chlamtac et al. 1979]), static delays (Aloha [Binder et al. 1975]), and dynamic backoff (Ethernet [Metcalfe & Boggs 1976]); we discuss the performance of these methods when applied to spin-waiting.

We propose a novel method for explicitly queueing spinning processors. As processors arrive at a lock, they each acquire a unique sequence number specifying the order that they will execute the critical section. When the lock is released, control can be directly passed to the next processor in line with no further synchronization and minimal effect on other processors.

We also examine the performance of several hardware solutions. We propose an addition to snoopy cache protocols that exploits the semantics of spin lock requests to obtain better performance.

The remainder of this paper discusses these issues in more detail. Section 2 outlines the range of architectures that we will consider and how these systems commonly support mutual exclusion. Section 3 analyzes the performance problems of simple software spin-waiting. Section 4 presents new software alternatives; Section 5 considers hardware solutions. Section 6 summarizes our conclusions.

2. Range of Multiprocessor Architectures Considered

While spinning processors can slow busy processors on any multiprocessor where spin-waiting consumes communication bandwidth, the precise performance of spin-waiting varies along several architectural dimensions: how processors are connected to memory, whether or not each processor has a hardware-managed coherent private cache, and if so, the coherence protocol. This paper will consider six types of architectures from within this design space:

- multistage interconnection network without coherent private caches
- multistage interconnection network with invalidation-based cache coherence using remote directories
- bus without coherent private caches
- bus with snoopy write-through invalidation-based cache coherence
- bus with snoopy write-back invalidation-based cache coherence
- bus with snoopy distributed-write cache coherence

(We assume for all of these that processors block when making a read request to memory.) While there are clearly some shared-memory architectures that are not represented in this list, these sample architectures expose most of the interesting issues in the performance of spin-waiting.

2.1. Common hardware support for mutual exclusion

Most architectures support mutual exclusion by providing instructions that atomically read, modify, and write memory. These atomic instructions are straightforward to implement. Conceptually, they require four services that might need inter-processor communication: the read and write, some method of arbitration between simultaneous requests, and some state that prevents further accesses from being granted while the instruction is being executed. Most multiprocessors are able to collapse these services into one or two bus or network transactions.

Multistage networks connect multiple processors to multiple memory modules. Memory requests are forwarded through a series of switches to the correct memory module. When a value is read from memory as part of an atomic instruction, any cached copies of the location (recorded in the directory associated with the memory module) must be invalidated and subsequent accesses to that memory module or at least to that location must be delayed (or refused and retried) while the new value is being computed. To minimize this delay, the computation can be done remotely by an ALU attached to each memory module. The Butterfly [BBN 1985] and RP3 [Pfister et al. 1985] implement this kind of remote "fetch and op".

In single bus multiprocessors, the bus can be used for arbitration between simultaneous atomic instructions. Before starting an atomic instruction, a processor acquires the bus and raises a line (*the atomic bus line*). This line is held while the new memory value is being computed to prevent further atomic requests from being started, but the bus can be released to allow other normal memory requests to proceed. Waiting atomic requests delay and only re-arbitrate for the bus when the line is dropped.

In systems that do not cache shared data, the bus transaction used to acquire the atomic bus line can be overlapped with the read request for the data. Similarly, with invalidation-based coherence [Archibald & Baer 1986], even if the lock value is cached, acquiring the atomic bus line can be overlapped with the signal to invalidate other cache copies. Note that normally the invalidation occurs even if the instruction does not change the value of the location, because it is done before the instruction executes.

Write-back invalidation-based coherence avoids an extra bus transaction to write the data. In this protocol, the new value is temporarily stored in the processor's cache. When another processor needs the value (for instance, as part of an atomic instruction), it gets the value at the same time it invalidates the first processor's copy.

With distributed-write write-back coherence, the initial read is usually not needed. Because copies in all caches are updated instead of invalidated when a processor changes a memory value, the cache block needed by an atomic instruction will often already be in the cache. In this case it would be wasteful of bus cycles to piggy-back the arbitration mechanism for the atomic bus line on top of the arbitration for the bus. For this reason, the Firefly, which implements distributed-write cache coherence, has a separate arbitration mechanism for its atomic bus line [Thacker et al. 1988]. A bus cycle is still usually needed at the end of the atomic instruction to update copies in other caches.

3. The Performance of Simple Approaches to Spin-Waiting

Given atomic read-modify-write instructions, it is relatively straightforward to develop a correct spin lock. For instance, each processor can execute an atomic test-and-set instruction to acquire the lock; this instruction reads the old value of the lock and sets it to busy. If the read returns that the lock was free, the processor has the lock; if the lock was busy, the processor must try again. The lock is released by (atomically) clearing the lock value.

It is more difficult to devise an efficient spin lock; this requires balancing several apparently opposing concerns. Performance when there is contention for the lock depends on minimizing the communication bandwidth used by spinning processors, since this can slow processors doing useful work; the delay between when a lock is released and when it is re-acquired by a spinning processor must also be minimized, since no processor is executing the critical section during this time. This appears to pose a tradeoff: the more frequently a processor tries to acquire a lock, the faster it will be acquired, but the more other processors will be disrupted.

Latency, the time for a processor to acquire a lock in the absence of contention, is also important, for instance to applications with frequent locking, yet containing no bottleneck lock. A complex algorithm that reduces the cost of spin-waiting could degrade overall performance if it takes longer to acquire the lock when there is no contention.

It might seem that the behavior of multiprocessors when there is contention for a spin lock is not important. A highly parallel application will by definition have no lock with significant amounts of contention, since that would imply a sequential component. If an application has a lock that is a bottleneck, the best alternative would be to redesign the application's algorithms to eliminate the contention. In no case does it make sense to add processors to an application if they end up only spin-waiting.

There are, however, several situations where spin lock performance when there is contention is important. Poor contention performance may prevent an application with a heavily utilized lock from reaching its peak performance, because the average number of spin-waiting processors will become non-trivial as the lock approaches saturation. Further, if processors arrive at a lock in a burst, queue lengths can be temporarily long, resulting in bad short-term performance, without the lock being a long-term bottleneck.

Alternately, it may not always be possible to tune a program to use the optimal number of processors An operating system, for instance, has little control over the rate at which users make operating system calls. At high load, locks that are normally not a problem could become sources of contention. Similarly, on a multiprogrammed multiprocessor, a naive user can inadvertently ruin performance for all other users by combining a bottleneck critical section, lots of processors, and an inefficient spin lock.

In this section, we analyze the performance of two simple spin-waiting algorithms; combined measurement results are presented at the end of the section.

3.1. Spin on test-and-set

The simplest spin-waiting algorithm is for each processor to repeatedly execute a test-and-set instruction until it succeeds at acquiring the lock. Table 3.1 lists sample code for this approach. Not surprisingly, the performance of spinning on test-and-set degrades badly as the number of spinning processors increases.

Init	`lock := CLEAR;`
Lock	`while (TestAndSet(lock) = BUSY)` `;`
Unlock	`lock := CLEAR;`

Table 3.1: Spin on Test-and-Set

Two factors cause this degradation. First, in order to release the lock, the lock holder must contend with spinning processors for exclusive access to the lock location. Most multiprocessor architectures have no way of giving priority to the clear request of the lock holder, requiring it to wait behind test-and-sets of spinning processors, even though these cannot succeed until the lock is released.

Further, on architectures where test-and-set requests share the same bus or network as normal memory references, the requests of spinning processors can slow accesses to other locations by the lock holder or by other busy processors. On multistage network architectures, spin-waiting can cause a "hot-spot", delaying accesses to the memory module containing the lock location as well as to other modules [Pfister & Norton 1985]. On bus-structured multiprocessors, each test-and-set consumes at least one bus transaction, regardless of whether the lock value is changed; these can saturate the bus.

3.2. Spin on read (test-and-test-and-set)

Intuitively, coherent caches should be able to reduce the cost of spin-waiting. Segall and Rudolph [1984] propose that spinning processors loop reading the value of the lock, and only when the lock is free, execute a test-and-set instruction; this eliminates the need to repeatedly test-and-set while the lock is held. They call this spinning on test-and-test-and-set; code for it is listed in Table 3.2. (We assume that boolean expressions are evaluated only if needed; the test-and-set is only executed if the lock is not busy.)

While the lock is busy, spinning is done in the cache without consuming bus or network cycles. When the lock is released, each copy is updated to the new value (distributed-write) or invalidated, causing a cache read miss that obtains the new value. The waiting processor sees the change in state and performs a test-and-set; if someone acquired the lock in the interim, the processor can resume spinning in its cache.

Lock

```
while (lock = BUSY or TestAndSet(lock) = BUSY)
    ;
```

Table 3.2: Spin on read (test-and-test-and-set)

When the critical section is small, however, spinning on a read has almost as much effect on busy processors as spinning directly on a test-and-set instruction. The reason is that transient behavior can dominate; when the lock is released and re-acquired by one of the waiting processors, it takes some time for the remaining processors to resume looping in their caches. During this time, most spinning processors have pending memory requests, delaying requests by busy processors during this interim. This behavior is most pronounced for systems with invalidation-based cache coherence, but it also occurs with distributed-write.

Suppose a number of processors are spinning reading the lock value in their caches. When the lock is released, these cache copies will all be invalidated; each processor will then incur a read miss to fetch the new value back into its cache. These read misses will be satisfied serially. Each processor to get the new value will then try to execute a test-and-set; these requests must compete for the bus or memory module with any remaining processors doing read misses.

The first processor to test-and-set will acquire the lock. Any processor who completed its read miss before this, however, will have seen the lock as free, proceed to do a test-and-set itself, fail, and go back to spinning reading the lock value. Unfortunately, each failing test-and-set instruction, because it is treated as a memory write, invalidates all cache copies of the lock, forcing any processors that had resumed spinning to miss again.

Thus, once the lock has been re-acquired, some processors have passed the barrier and have a pending test-and-set request; the remainder have pending reads, trying to fill their cache after the original read miss. (The number of processors who have seen the lock as free will be worse on systems with multistage networks, because of the greater distance between the processors and memory.) Each read miss that is satisfied decreases the number of pending requests; that processor obtains a cache copy of the lock and resumes looping. Each test-and-set request that is satisfied decreases the number processors waiting to test-and-set; however, it also invalidates all existing cache copies of the lock, forcing those processors that had been spinning in their cache to read miss again. After each test-and-set, every processor but the one that did the test-and-set must contend for memory. Eventually, the last spinning processor does a test-and-set, allowing every other spinning processor to do a read miss and then quiesce.

Before quiescence, each spinning processor spends most of its time contending for the bus or memory. After quiescence, spinning processors consume no communication resources. Thus, a normal memory request will be slowed dramatically if it occurs before quiescence and not at all if it occurs afterwards. For long critical sections, this initial slowdown is less significant, but for short critical sections, it dominates performance.

Our discussion so far has assumed random arbitration among memory requests. It might seem that spinning on a cache copy would perform well given fixed priority bus arbitration, as for instance on the Firefly. When a lock is released, the highest priority processor will acquire the lock. Even if it takes some time for the other processors to quiesce, the lock holder would not be slowed since it has higher priority than the other processors. However, if the lock is released before quiescence, a low priority processor with a pending test-and-set could acquire the lock before higher priority processors looping on read. The lock holder might then be delayed by these higher priority processors.

3.3. Reasons for the poor performance of spin on read

There are several factors that cause the performance of spinning on a memory read to be worse than expected.

- There is a separation between detecting that the lock has been released and attempting to acquire it with a test-and-set instruction. This separation allows more than one processor to notice that the lock has been released, pass by that test, and proceed to try a test-and-set. Ideally, if one processor could notice the change and acquire the lock before any other processor committed to doing a test-and-set, the performance would be better.
- Cache copies of the lock value are invalidated by a test-and-set instruction even if the value is not changed. If this were not the case, invalidations would occur only when the lock is released and then again when it is re-acquired.
- Invalidation-based cache-coherence requires $O(P)$ bus or network cycles to broadcast a value to P waiting processors. This occurs despite the fact that, after an invalidation, they each request exactly the same data.

While a solution to any of these three problems by itself would result in better performance, any single solution would still require bus activity that grows linearly with the number of processors.

For example, distributed-write cache coherence eliminates invalidations; each processor directly receives all updates to the lock value. All reads can therefore be done locally; only test-and-sets still require bus traffic. The Sequent Balance [Beck et al. 1987] and the Silicon Graphics 4D-MP [Baskett et al. 1988] both use a separate bus for test-and-set variables for just this reason; the bus implements distributed-write coherence to reduce bus traffic due to spin-waiting. Unfortunately, broadcasting updates makes the separation between the test and the test-and-set worse: all processors receive the updated lock value at the same time, and all therefore proceed to try the test-and-set. The result is that P test-and-sets must be performed before quiescence. It is unclear whether either the Balance or the 4D-MP has special hardware to avoid this problem.

3.4. Measurement results

To demonstrate the performance of simple spin-waiting, we implemented both approaches on a Sequent Symmetry Model B shared-memory multiprocessor with 20 80386 (approximately 2 MIP) processors. The Symmetry has a shared bus and write-back invalidation-based cache coherence; unlike the Balance, test-and-set variables are handled on the same bus as normal memory references [Lovett & Thakkar 1988]. Acquiring and releasing a lock on the Symmetry normally takes 5.6 microseconds, less if the cache block containing the lock is initially private to the locking processor.

Figure 3.1 is the principal performance comparison: the elapsed time for various number of processors to cooperatively execute a critical section 1 million times, for the two alternatives. Each processor loops: wait for the lock, do the critical section once, release the lock, and delay for a time randomly selected from a uniform distribution. The mean delay is equal to five times the size of the critical section. The wait in the loop eliminates any locality effect: each iteration, the lock and the shared data accessed by the

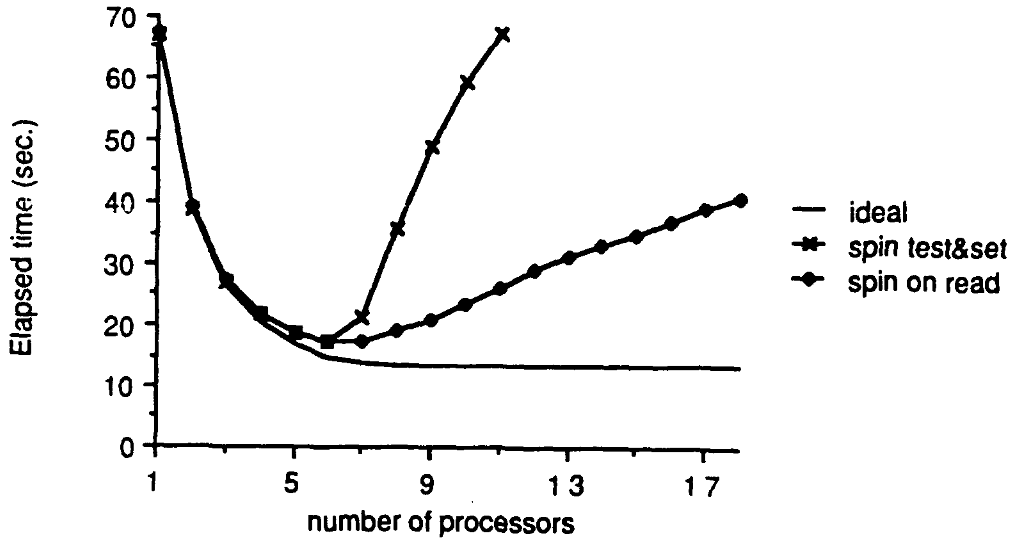


Figure 3.1: Principal performance comparison: Elapsed time (sec.) to execute benchmark (measured)

Each processor loops 1 million / P times: acquire lock, do critical section, release lock, and compute.

critical section move between caches. The lock and shared data are placed so as to fall in separate cache blocks.

This benchmark simulates the performance of an application with a small central critical section. (Similar curves have been measured using a fixed delay between lock accesses.) It also shows spin lock latency and performance with small and large amounts of contention. Ideally, performance initially improves as processors are added, due to increased parallelism, but as the critical section becomes a bottleneck, performance levels out. The ideal curve in Figure 3.1 is the time the test would have taken, given free spin-waiting; this was determined by simulation from the time to execute the critical section and the mean delay between lock accesses.

Figure 3.1 confirms our analysis. Performance degrades badly as processors spin on test-and-set; spinning on a read is better, but it still has disappointing performance. As the critical section becomes a bottleneck, the average number of spin-waiting processors increases, significantly slowing the processor executing the critical section. As a result, peak ideal performance is never reached. Performance with these alternatives is very sensitive to the exact number of processors given to an application; adding even a few processors beyond where the lock saturates worsens overall performance considerably.

This behavior can be degenerative [Anderson et al. 1989]. Critical sections, since their purpose is to manipulate shared data structures, typically have higher memory access rates than non-critical sections. As a critical section becomes a bottleneck, the spinning processors slow the lock holder's execution, both in absolute terms and relative to non-critical sections, making it more of a bottleneck, resulting in more spinning processors.

As we noted, there is a difference in the effect on memory accesses before and after quiescence when processors spin on a read. This two-phase behavior allows us to measure the time to quiesce on the Symmetry. We construct a critical section whose behavior mirrors that of the bus, but in reverse. The critical section begins by delaying for some amount of time without using the bus at all, then proceeds to use the bus heavily before releasing the lock. If the initial delay is longer than the time to quiesce the spinning processors, then the critical section will run as fast on *P* processors as on one. If the heavy bus usage begins before quiescence, the critical section will run slower on *P* processors. We vary the length of the initial delay to find this performance knee; in practice, this knee was quite sharp.

Figure 3.2 shows the results of this test. The time to quiesce grows steeply but linearly with the number of processors. As a result, even a few spinning processors can adversely impact the execution speed of a moderate-sized critical section.

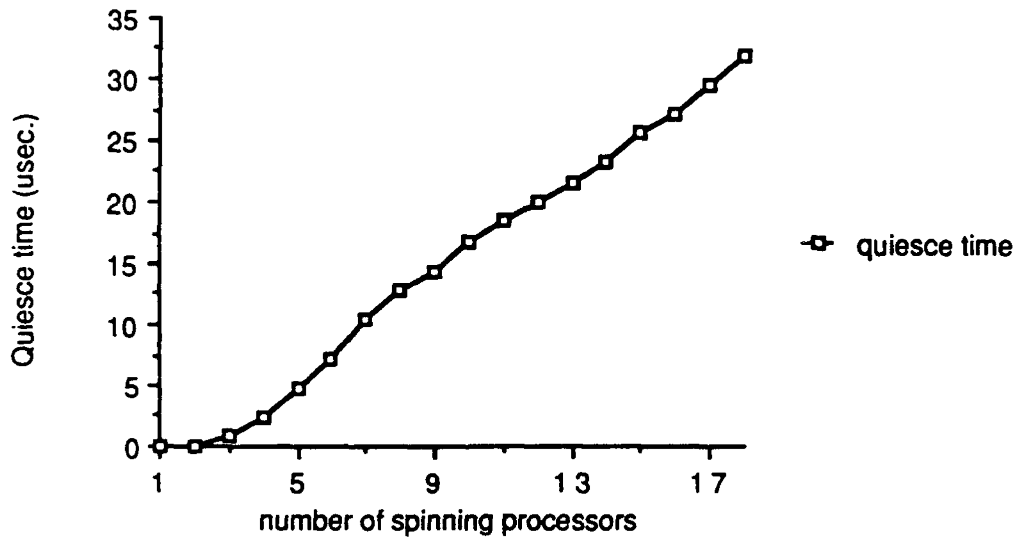


Figure 3.2: Time to quiesce, spin on read (μsec.)

4. New Software Alternatives

In this section, we first describe five software spin-waiting approaches, four based on CSMA network protocols and one using explicit queueing, leaving until afterwards the presentation of their combined measurement results.

4.1. Delay alternatives

We consider four ways of inserting delays into the spin-wait loop, defined by two dimensions: where the delay is inserted and whether the size of the delay is set statically or dynamically. A delay can be inserted after the lock has been released or alternatively after every separate access to the lock; code for these approaches is listed in Tables 4.1 and 4.2. Because processors first try to acquire the lock before delaying, lock latency is unaffected.

Lock

```
while (lock = BUSY or TestAndSet(lock) = BUSY)
     begin
     while (lock = BUSY) ;
     Delay();
     end;
```

Table 4.1: Delay After Spinner Notices Released Lock

Lock

```
while (lock = BUSY or TestAndSet(lock) = BUSY)
     Delay();
```

Table 4.2: Delay Between Each Reference

4.1.1. Delay after spinning processor notices lock has been released

We can reduce the number of unsuccessful test-and-sets when spinning on a read by inserting a delay between when a processor reads that the lock is released and when it commits to trying the test-and-set. If some other processor acquires the lock during this delay, then the processor can resume spinning; if not, then the processor can try the test-and-set, with a greater likelihood that the lock will be acquired. In this way, the number of unsuccessful test-and-sets, and thus invalidations, can be reduced.

Each processor can be statically assigned a separate slot, or amount of time to delay, from 0 to $P - 1$, where P is the number of processors. The spinning processor with the smallest assigned delay checks the lock, sees that it is free, and acquires it. Processors with longer delays then time out, see that the lock is busy (enduring another cache miss), and resume spinning. By statically assigning delays, we can ensure that at most one processor times out at any instant. Chlamtac et al. [1979] propose a similar method to arbitrate access to a CSMA network. (Slots can also be used to implement priority access to the critical section, by assigning lower delays to higher priority processes.)

This algorithm performs well when there are many spinning processors. It is likely that some spinning processor will have a short delay; when the lock is released, some processor will quickly re-acquire it. When there is only one spinning processor, however, it is unlikely to have a short delay, leaving the lock unacquired for a relatively long time, harming performance.

The number of slots can be varied to trade off performance between these two cases. When there are few spinning processors, using fewer slots improves performance by reducing the time to pass control of the lock to a waiting processor. When there are many spinning processors, using fewer slots worsens performance since more than one processor would simultaneously time out and attempt to test-and-set, requiring longer to quiesce.

By varying spinning behavior based on the number of waiting processors, we can have good performance in both situations. Such an algorithm has already been devised for CSMA networks: Ethernet's exponential backoff [Metcalfe & Boggs 1976]. In a CSMA network, each processor can detect when the network is being used. When the network is unused, a processor can acquire the network by beginning to transmit, but if another processor simultaneously begins transmitting ("collides"), they both fail and must retry. The idea is for each processor to use the number of collisions it has experienced to estimate the number of spinning processors.

Initially, an arriving processor assumes that there are no other processors waiting to use the network and chooses a random delay with a small mean. Whenever it times out, tries to acquire the network and fails because some other processor timed out at the same time, then, assuming random arrivals, there are likely to be many more waiting processors that did not collide. Collisions are unlikely if the average delay is at least half the number of spinning processors. In Ethernet, then, each processor doubles its mean delay after each collision.

Analogously, a processor trying to acquire a spin lock could begin by assuming there were no other waiting processors. Each time it times out, sees the lock is still free, tries to test-and-set and fails, it has "collided" with at least one other processor. There are likely to be many other spinning processors it did not collide with, and thus it should double its mean delay, up to some limit.

Although Ethernet's backoff has been shown to have good performance [Metcalfe & Boggs 1976], the performance of backoff for spin locks will not be the same as for networks. In a network, a collision aborts all processors; there is an equal cost to a collision among any number of processors as there is to an empty slot. By contrast, a test-and-set collision allows one processor to proceed, and the cost depends on how many processors collide. The more processors that try to acquire the lock and fail, the longer it will take them to quiesce, and the more that other processors, including the lock holder, will be slowed.

In designing a backoff scheme for spin locks, there are a number of details that affect performance. Our first implementation got most of these wrong.

- When a processor detects that the lock has been acquired, it should not increase (or decrease) its mean delay. The fact that some other processor had a shorter delay does not imply much about how many other spinning processors there are.
- There needs to be a maximum bound on the mean delay. Otherwise, if a processor backs off a number of times and then becomes the only waiting processor, it will take a long time for it to acquire the lock. This bound should be equal to the number of processors, so that backoff has the same performance as statically assigned slots when there are many spinning processors.
- The initial delay of an arriving processor should be some fraction of its delay the last time at the lock. In a CSMA network, an arriving processor can efficiently re-estimate the number of spinning processors because collisions are not unduly costly. For spin locks, however, the learning curve can be expensive. There is no more reason to assume initially that there are no other spinning processors than that the number is related to past experience. For our measurements, we set the initial delay to be half the previous delay. Note that in Table 4.1 if the lock is free when the processor arrives, it will

immediately acquire it; backoff is only used if the lock is initially busy.

While the justification for backoff assumes random arrivals at the lock, it performs well compared to using static slots even when this is not the case. If processors execute for a fixed amount of time between lock accesses, they will tend to self-schedule so that either there is no contention for the lock or there are always the same number of spinning processors. In the latter case, backoff would increase the delays until there were few collisions, and then the hysteresis would help maintain those delays.

Similarly, both backoff and static slots have performance problems when processors repeatedly arrive at a lock in a burst. The first time the lock is accessed, all processors choose a small delay, time out together, and take a long time to quiesce. Eventually the mean delays are increased enough to avoid collisions; this initial degradation is largely avoided the next iteration. Using static slots also avoids this initial performance degradation. However, since the number of waiting processors decreases as more acquire the lock, both alternatives are finally left with processors with inappropriately long delays, making it take longer to pass control of the lock.

Polling for the lock release is only practical for systems with per-processor coherent caches. On other systems, processors would consume communication bandwidth if they were to spin reading memory waiting for the lock to be released. Some multistage network multiprocessors with caches based on remote directories limit the number of outstanding copies of a location in order to limit the size of the directories [Agarwal et al. 1988]; if the number of spinning processors exceeds this number, spinning on a read degenerates to spinning across the network.

Even for multiprocessors with snoopy or complete directory invalidation-based caches, using exponential backoff or static slots solves only one of the problems with spinning on a memory read. Each spinning processor still requires at least two cache read misses per execution of the critical section, one when the lock is released and one when the lock is acquired. For sufficient numbers of spinning processors, this read miss activity can saturate the bus or network.

Exponential backoff after the lock is released does, however, provide scalable performance for multiprocessors with distributed-write cache coherence. In these systems, each test-and-set requires a single bus cycle to broadcast the new value, independent of the number of spinning processors. Exponential backoff, in turn, limits the number of unsuccessful test-and-sets.

4.1.2. Delay between each memory reference

An alternative approach to reducing the cost of spin-waiting is to insert a delay between each memory reference. This can be used on architectures without coherent caches or with invalidation-based coherence to limit the communication bandwidth consumed by spinning processors. In the code in Table 4.2, we check if the lock is free before trying to test-and-set since we assume that a test-and-set instruction consumes more bandwidth than a simple read.

The mean delay between each reference can be set statically or dynamically, analogous to the Aloha [Binder et al. 1975] and Ethernet network protocols. Most of the tradeoffs outlined above apply to these alternatives: more frequent polling improves performance when there are few spinning processors and worsens performance when there are many. Exponential backoff can be used to dynamically adapt to varying conditions.

Delaying between each reference poses special problems, however. For instance, the performance of backoff is bad when there is a single spinning processor for a moderate-sized critical section. The processor will continue to back off the delay as long as the lock is held. When the lock is released, the spinning processor will be in the midst of a long delay that must finish before it notices the change.

4.2. Queueing in shared memory

It might seem that shared memory could be used to store state to control the activity of spinning processors. This is less easy than it appears. For instance, a shared counter could be used to directly keep track of the number of spinning processors, instead of relying on a backoff algorithm to estimate that number. Given atomic increment and decrement instructions, the apparent cost of maintaining this state is the execution of two extra atomic instructions per critical section. However, each spinning processor must read this data to compute its delay; on systems without distributed-write coherence, this would consume as much bandwidth as directly polling the lock.

Another approach would be to maintain an explicit queue of spinning processors. Each arriving processor enqueues itself and then spins on a separate flag. When the processor finishes with the critical section, it dequeues itself and sets the flag of the next processor in the queue. This approach can reduce invalidations: if each processor's flag is kept in a separate cache block, then only one cache read miss is needed to notify the next processor. Maintaining queues, however, is expensive; the enqueue and dequeue operations must themselves be locked. (Even if there are atomic enqueue and dequeue instructions, as on the VAX, these operations are likely to be slow since they must modify more than one location.) The result is much worse performance for small critical sections. For instance, it would not be reasonable to do this if the critical section itself was a queue operation.

We have developed a method of queueing spin-waiting processors that requires only a single atomic operation per execution of the critical section. Each arriving processor does an atomic read-and-increment to obtain a unique sequence number. When a processor finishes with the lock, it taps the processor with the next highest sequence number; that processor now owns the lock. Since processors are sequenced, no atomic read-modify-write instruction is needed to pass control of the lock. Table 4.3 lists the code for this approach ("myPlace" is a location private to each processor). Sequent [Graunke 1988] has independently devised a similar algorithm.

Init

```
flags[0] := HAS_LOCK;
flags[1..P-1] := MUST_WAIT;
queueLast := 0;
```

Lock

```
myPlace := ReadAndIncrement(queueLast);
while (flags[myPlace mod P] = MUST_WAIT)
    ;
flags[myPlace mod P] := MUST_WAIT;
```

Unlock

```
flags[(myPlace + 1) mod P] := HAS_LOCK;
```

Table 4.3: Queue Using Atomic Read-and-Increment

The best implementation varies somewhat among architectures. With distributed-write coherence, processors can all spin on a single counter. To release the lock, a processor simply writes its sequence number into the counter; each processor's cache is updated, directly notifying the next processor in line with a single bus transaction.

With invalidation-based coherence, each processor should wait on a flag in a separate cache block. Only two bus or network transactions (an invalidation and a read miss) are needed to signal the next processor. Similarly, on a multistage network without coherent caches, each flag should be placed in a separate memory module. Even though processors must poll to learn when it is their turn, there can be no more than P such polling requests outstanding at a time among $P \times \log P$ switches and P memory modules.

This approach is less valuable in a system with a bus but no cache coherence. Processors must still poll to find out if it is their turn; the bus can easily be swamped with this polling. To be effective, a delay can be inserted between each poll that depends on how close the processor is to the front of the queue and on how long it takes to execute the critical section. This indicates one way of using fewer than P separate memory locations in Table 4.3: if a processor is farther from the front than the number of flags, it can poll (rarely) to find out when it is close enough to spin on its own flag.

If an architecture does not support an atomic read-and-increment instruction, this operation can itself be locked. Since the operation would take at most a few instructions, it would not normally become a bottleneck except when used with the most trivial of critical sections. When processors arrive in a burst, however, there can be short-term contention for this lock. In this case, one of the delay alternatives from Section 4.1 should be chosen to minimize bus traffic. The tradeoffs are slightly different here; a delay in passing control over the read-and-increment operation need not impact overall performance, provided some backlog of spinning processors have already obtained a number. Interestingly, the Symmetry supports an atomic increment but not an atomic read-and-increment instruction (the original value is not saved).

Because a spinning processor automatically gets control of the critical section when its bit is set, the time between when one processor finishes and the next processor starts executing the critical section is reduced. In some sense, this exploits parallelism: the spinning processor does the time-consuming work of the atomic operation before the lock is released, decreasing the amount of serial work required to pass control. Thus, throughput actually increases as a critical section becomes a bottleneck.

Unfortunately, queueing has some bad aspects. It increases lock latency. Each processor must increment a counter, check a location, zero that location, and set another location; in the other methods, when there is no contention, the first test-and-set acquires the lock. Thus, when there is contention, queueing is better; when there is no contention, backoff or simple spin-waiting is better.

While processor preemption can yield bad spin-locking performance [Zahorjan et al. 1988], queueing makes this problem more severe. Normally, if a process holding a lock is preempted, every process spinning on that lock must wait for it to be re-scheduled. Good performance requires lock holders to not be preempted. With queueing, however, preempting any spin-waiting process forces all behind it to wait if it reaches the front of the queue before being re-scheduled. This can cause lock-step behavior if a small critical section is accessed frequently. When a process in line is preempted, other processes queue up behind it; when it is re-scheduled, it uses the lock once, but may then have to wait when it re-accesses the lock for other processes that have been preempted in the interim. One way of avoiding this problem, if a process can be notified before it is preempted, is for it to remove itself from the queue by notifying the next process in line of that event (e.g., by setting a bit).

Another problem with queueing is that it makes it more difficult to wait for multiple events. As the number of processors increases, any centralized resource can become a bottleneck. One way of increasing throughput is to divide control over a resource among several critical sections, so that a spinning processor need access only one of the locks to get service. It is easy to see how delays could be used in this case; each waiting processor could randomly poll a server, and if busy, delay before polling another server. It is hard to see how queueing could be adapted, however, since a processor would only be able to wait in one queue at a time.

4.3. Measurement results

We implemented the five software alternatives we have described on the Symmetry Model B with 20 processors. The static and dynamic delays varied from 0 to 15 microseconds; it takes approximately one microsecond on the Symmetry to execute the test-and-set instruction. Since the Symmetry does not support an atomic read-and-increment instruction, queueing uses an explicit lock (with backoff after each memory reference) to access the sequence number.

Figure 4.1 is the principal performance comparison: spin-waiting overhead to execute the benchmark used for Figure 3.1, as a function of the number of processors. To isolate the effect of spinning, we subtract from the elapsed time to execute the benchmark the "ideal" curve, the time the test would have taken given free spin-waiting. This leaves just the component due to spin-waiting overhead. We include spin on read for comparison.

Figure 4.1 confirms our analysis. Although performance varies, all five methods described in this section have reasonable performance across the range of conditions. The one processor time reflects lock latency; queueing has high latency, while all other alternatives have low latency. Queueing would have better latency on systems with an atomic read-and-increment instruction. As the lock approaches saturation, the static delay alternatives have worse performance, because the delays are inappropriate for small numbers of spinning processors. The performance of the backoff alternatives remains close to the simple spin-waiting methods by adapting to the number of spinning processors.

When there are high numbers of spinning processors, backoff performs slightly worse than static delays; some collisions are necessary to maintain appropriate delays. Queueing performs best in this case by parallelizing the lock handoff. Across the entire spectrum, because of the Symmetry's invalidation-based coherence, delaying after each reference is slightly better than delaying after the lock is released.

To demonstrate the potential benefit of backoff relative to static delays, Figure 4.2 compares spin-waiting overhead for the benchmark as a function of the number of static slots. As can be seen, small

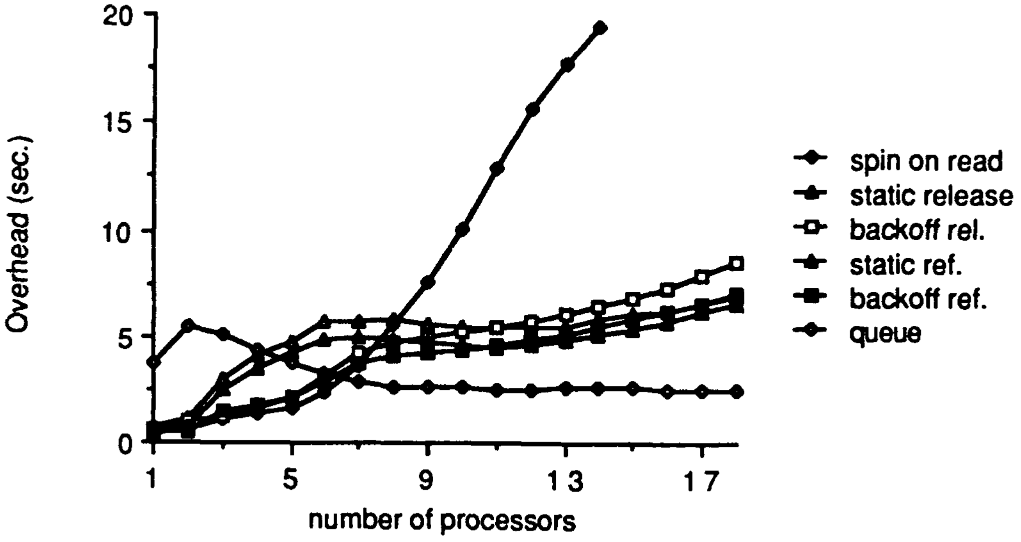


Figure 4.1: Principal performance comparison:
Spin-waiting overhead (sec.) in executing the benchmark (measured)

Each processor loops 1 million / P times: acquire lock, do critical section, release lock, and compute.

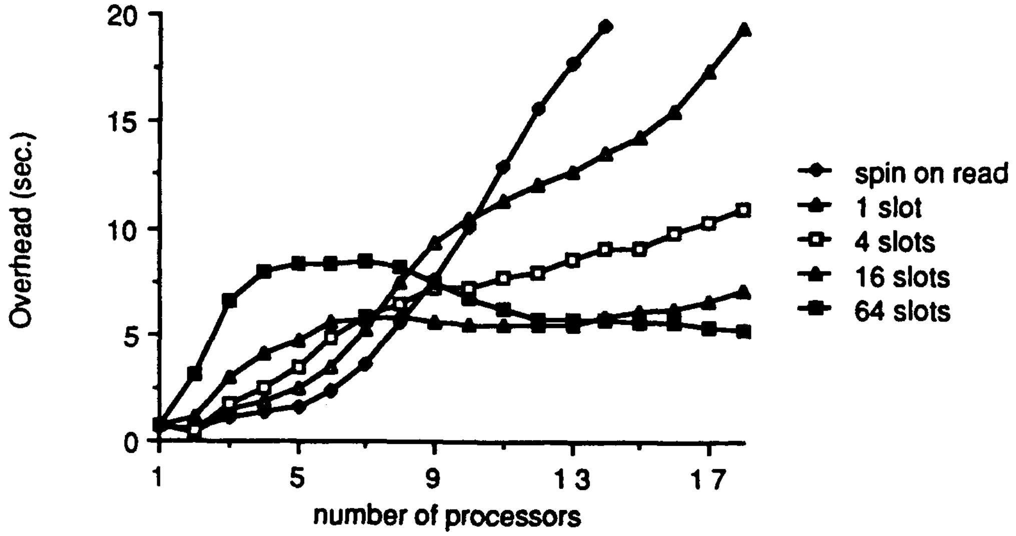


Figure 4.2: Spin-waiting overhead (sec.) vs. number of slots

numbers of slots perform better when there are few spinning processors, while larger numbers of slots perform better when there are many. This tradeoff becomes harsher as the maximum number of spinning processors increases: 64 slots has much worse low load performance than direct spinning on read, yet that number of slots might be necessary to avoid poor high load performance in systems with large numbers of processors. Backoff avoids the tradeoff by performing well in both situations.

Figure 4.3 shows spin-waiting overhead when processors arrive at a spin lock at the same time. A timestamp is taken before the processors are released from a barrier; each processor then acquires the lock and bumps a counter; and another timestamp is taken when the last processor acquires the lock. As in Figure 4.1, we subtract the time to execute this test given free spin-waiting. This result is then normalized by the number of processors, to yield the average spin-waiting overhead per execution of the critical section. For clarity, we omit the curves for static and dynamic delays after the lock is released, as these are everywhere slightly worse than delaying between each reference.

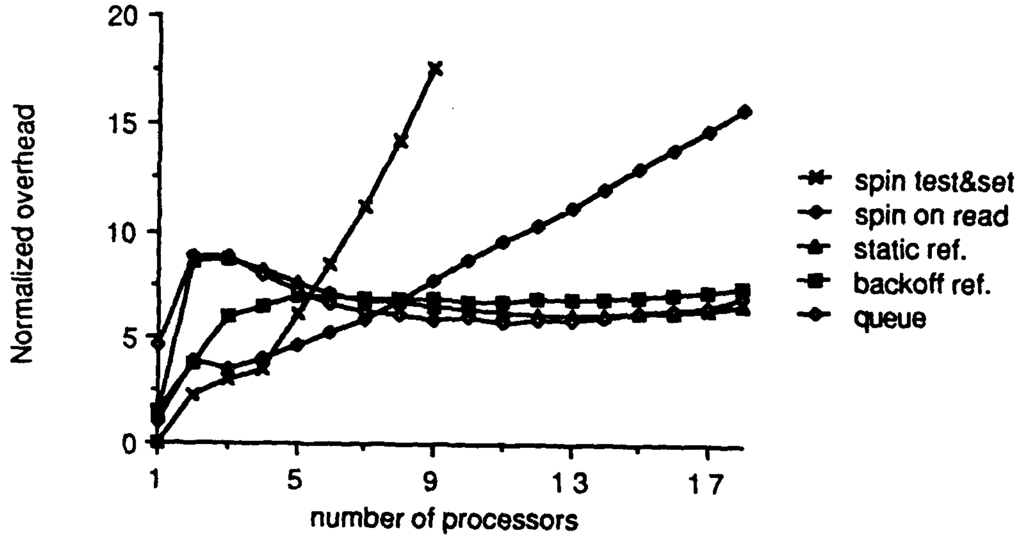


Figure 4.3: Spin-waiting overhead in achieving barrier, normalized by the number of processors (µsec. per processor)

The results in Figure 4.3 are similar to that of Figure 4.1. Queueing has bad latency in the one processor case. When two processors arrive together, using a static mean delay performs worst, but all alternatives perform badly because of the initial contention. As the number of processors increases, the behavior becomes similar to that of Figure 4.1, except that queuing does not perform well with high numbers of processors because it uses backoff to arbitrate for the lock protecting its sequence number.

5. Hardware Solutions

In this section, we consider hardware changes to improve spin lock performance. As for the software alternatives, implementing solutions in hardware also poses tradeoffs. For example, the best cache coherence mechanism for spin locks may not be the best for normal memory references; some systems, such as the Balance and the 4D-MP, try to avoid this dilemma by using one bus with invalidation-based coherence for normal requests and a separate one with distributed-write coherence for test-and-set variables. Unfortunately, this duplication adds expense that is of little benefit to applications that do not spend significant amounts of time spin-waiting. Further, if this separate bus is slower than the normal bus, as on the Balance, lock latency will suffer.

We consider the question of hardware solutions separately for multistage network and single bus multiprocessors.

5.1. Multistage interconnection network multiprocessors

Combining networks, by providing parallel access to a single memory location [Pfister & Norton 1985], can improve the performance of spinning directly on test-and-set. Requests to the same location that arrive at the same network switch are combined and forwarded as a single request; the result is the same as if the two requests were made sequentially at the memory module. For example, two test-and-set requests would result in one request being forwarded and one request returning immediately with the value as set; no matter what the current value, only one will succeed if the two requests are made sequentially. Similarly, a test-and-set and a clear (to release the lock) would be combined to forward the set, while the test-and-set request returns having acquired the lock.

Assuming the cycle time of the combining network is the same as a normal network, combining has good performance for any number of spinning processors. When there is no or little contention, there is little combining, and performance is similar to normal spinning on test-and-set. As more processors spin-wait, combining reduces congestion due to duplicate test-and-sets, and since the request to release the lock is likely to be combined with a test-and-set at an earlier stage of the network, the time to pass control of the lock would be reduced. However, since the complexity of combining switches is likely to increase their latency, better performance might be obtained by a normal network with backoff or queueing.

Hardware queueing at the memory module, like software queueing, can eliminate polling across the network; it can also speed passing control of the lock. For this, processors would issue explicit "enter" and "exit" critical section instructions to the memory module, which would maintain queues of the processors waiting for each lock. When a processor's "enter" request returns, it has the lock; no polling across the network is necessary. With software queueing on a system with coherent caches, the processor releasing the lock notifies the next processor by writing its flag; an invalidation followed by a read miss is needed before the spinning processor can start executing the critical section. By specially handling critical section requests, hardware queueing eliminates one network round trip to pass control of the lock. Perhaps most importantly, lock latency is likely to be better with hardware than with software queueing; even though hardware queueing increases complexity at the memory module, it reduces the number of instructions needed to acquire the lock.

Goodman et al. [1989], albeit for a different architecture, have proposed using caches to hold queue links. Their approach stores the name of the next processor in the queue directly in each processor's cache; when the lock is released, the next processor can be notified without going through the original memory module. To enhance flexibility, they have also proposed that control return to software after the processor is put on the queue for a critical section; the processor is then separately notified by the hardware when it gets to the front of the queue.

5.2. Single bus multiprocessors

One obvious solution to reducing the number of invalidations caused by spinning on a read would be to invalidate only if the lock value changes. Before starting an atomic instruction, a processor would acquire the bus and raise a line to to prevent other processors from accessing their potentially incorrect cache copies. These copies would then be invalidated only if the value changes. Unfortunately, this solves only one of the problems with spinning on a read. When the lock is released, there will still an invalidation, a cache miss by each spinning processor, followed by some number of failing test-and-sets; each of these consumes bus bandwidth. The time to quiesce is reduced but not eliminated. Unlike software queueing or backoff, performance degrades as more processors spin.

Rather, we note that more intelligent snooping of bus activity can reduce the cost of spin-waiting. We have already seen this in practice. If hardware keeps caches coherent, processors can spin on a cache copy instead of repeatedly reading from memory. Similarly, invalidation-based coherence can result in a cascade of read misses, which do not occur given write-broadcast coherence.

We will present two ways of improving performance by using information transmitted over the bus. One eliminates duplicate read requests; the other eliminates redundant test-and-sets. Simple spin-waiting is expensive because all spinning processors make bus requests to do the same thing, read or test-and-set, at the same time. This fact can be used to advantage.

Read broadcast [Segall & Rudolph 1984; Karlin et al. 1986] can eliminate duplicate read miss requests. Each processor's cache controller monitors the bus; if a read occurs corresponding to an invalid block in its cache, it takes the data off the bus and sets the block to valid. Thus, whenever the cache copies of spinning processors are invalidated, the first read will fill all caches. Some spinning processors, however, will have already seen the cache as invalid and will be waiting at the bus to do the read; if a controller with a pending read observes the bus grant a read on the same location to some other processor, it should simply wait and take the data returning for that request. This eliminates the cascade of read misses when spinning on a read, without implementing full distributed-write coherence.

By specially handling test-and-set requests in the cache and bus controllers, we can eliminate the need for failing test-and-sets to use the bus. This way, processors can spin on test-and-set, acquiring the lock quickly when it is free, without consuming bus bandwidth when the lock is busy. Provided that specially handling test-and-sets does not increase the bus or cache cycle time, its performance would be better than software backoff or queueing. Figure 4.1 shows that neither of these achieves ideal performance on the Symmetry. As the critical section becomes a bottleneck, backoff performance degrades slightly because of the overhead of computing random delays; the complexity of queueing similarly increases lock latency.

The idea is to not commit to doing the test-and-set over the bus so long as there is the possibility that it might fail (return that the lock is busy), and to return immediately without using the bus whenever the test-and-set would fail if it were the next to execute.

When a processor issues a test-and-set request, it first checks the cache. If the lock is not in the cache (because it was replaced or invalidated), a read miss occurs. Duplicate read misses can be eliminated using read broadcast. Once the lock value is in the cache, the test-and-set can return immediately if the lock is busy. If the lock is free, the controller can then try to acquire the bus to get the mutual exclusion needed by the atomic instruction.

While the controller is waiting for the bus, it must monitor the bus activity to determine if it should continue waiting. With distributed-write coherence, if some other processor acquires the bus to do a test-and-set, it will broadcast the new lock value, and all pending test-and-set requests can be aborted. If the lock value is invalidated, the processor must convert the test-and-set request back to a read request to see if the lock is now busy.

Typically, cache and bus controllers do not know the type of atomic instruction making a request, since the ALU is responsible for performing the logic of the instruction. This information is needed for the cache to be able to abort pending test-and-sets. When the cache returns control to the processor, the processor can proceed as if it had exclusive access, whether or not the test-and-set actually acquired the bus. In one case, it really has the exclusive access needed to acquire the lock; in the other, it can proceed because its actions will be consistent with some serial ordering of atomic instructions.

6. Conclusions

In this paper, we have shown that simple methods of spin-waiting for mutually exclusive access to shared data structures degrade overall performance as the number of spinning processors increases. We have proposed and analyzed the performance of several hardware and software solutions to this problem.

For multiprocessors without special support for spin-waiting beyond implementing atomic instructions, we have shown that software queueing and a variant of Ethernet backoff have good performance even for large numbers of spinning processors. Because it is simpler, backoff has better performance when there is no contention for the lock; queueing, by parallelizing the lock handoff, performs best when there are waiting processors.

We have also shown that performance can be further improved by specially handling spin lock requests. On multiprocessors with multistage interconnection networks, explicit hardware queueing of spin-waiting processors, whether at the memory module or in each cache, can reduce the time to pass control of the lock to a waiting processor. On shared bus multiprocessors, failing test-and-sets can be handled with no bus traffic given more intelligent snooping. Whether real workloads will have significant enough amounts of spin-waiting to make such additional hardware support worthwhile remains an open question.

Acknowledgments

The author would like to thank Mark Donner, Jim Goodman, Dave Keppel, Ed Lazowska, Dave Wagner, John Zahorjan, and the referees for helpful discussions of the issues presented in this paper.

References

[Agarwal et al. 1988]
A. Agarwal, R. Simoni, J. Hennessey, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. *Proc. 15th International Symposium on Computer Architecture*, pp. 280-289, June, 1988.

[Agarwal & Cherian 1989]
A. Agarwal and M. Cherian. Adaptive Backoff Synchronization Techniques. *Proc. 16th International Symposium on Computer Architecture*, pp. 396-406, June, 1989.

[Anderson et al. 1988]
T. E. Anderson, E. D. Lazowska, and H. M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. *1989 ACM SIGMETRICS and Performance '89 Conference on Measurement and Modeling of Computer Systems*, pp. 49-60, May 1989.

[Archibald & Baer 1986]
J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, vol. 4, no. 4, Nov. 1986.

[Baskett et al. 1988]
F. Baskett, T. Jermoluk, and D. Solomon. The 4D-MP Graphics Superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100000 Lighted Polygons per Second. *IEEE Spring COMPCON*, pp. 468-471, 1988.

[BBN 1985]
BBN Laboratories. Butterfly Parallel Processor Overview. 1985.

[Beck et al. 1987]
B. Beck, B. Kasten, and S. Thakkar. VLSI Assist for a Multiprocessor. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pp. 10-20, Oct. 1987.

[Binder et al. 1975]
R. Binder, N. Abrahamson, F, Kuo, A. Okinawa, and D. Wax. Aloha Packet Broadcasting -- a Retrospective. *AFIPS Conference Proceedings*, 1975.

[Chlamtac et al. 1979]
I. Chlamtac, W. Franta, and D. Levin. BRAM: The Broadcast Recognizing Access Method. *IEEE Transactions on Communication*, vol. 27, pp. 1183-1190, Aug. 1987.

[Gajski et al. 1983]
D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. CEDAR -- A Large Scale Multiprocessor. *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 524-529, Aug. 1983.

[Goodman & Woest 1988]
J. Goodman and P. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 442-431, June 1988.

[Goodman et al. 1989]
J. Goodman, M. Vernon, and P. Woest. A Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, April 1989.

[Graunke 1988]
G. Graunke. Personal communication. 1988.

[Herlihy 1988]
M. Herlihy. Impossibility and Universality Results for Wait-free Synchronization. *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pp. 276-291, 1988.

[Jayasimha 1987]
D. N. Jayasimha. Parallel Access to Synchronization Variables. *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 97 - 100, Aug. 1987.

[Karlin et al. 1986]
A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive Snoopy Caching. *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pp. 244-254, Oct. 1986.

[Lamport 1987]
L. Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, vol. 5, no. 1, 1987.

[Lovett & Thakkar 1988]
T. Lovett and S. Thakkar. The Symmetry Multiprocessor System. *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 303-310, Aug. 1988.

[Metcalfe & Boggs 1976]
R. Metcalfe and D. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, vol. 19, no. 7, pp. 395-404, July 1976.

[Pfister & Norton 1985]
G. Pfister and V. Norton. "Hot-Spot" Contention and Combining in Multistage Interconnection Networks. *ACM Transactions on Computer Systems*, vol. 3, no. 4, Oct. 1985.

[Perron & Mundie 1986]
R. Perron and C. Mundie. The Architecture of the Alliant FX/8 Computer. *IEEE COMPCON*, 1986.

[Pfister et al. 1985]
G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weise. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[Segall & Rudolph 1984]
Z. Segall and L. Rudolph. Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor. *Proceedings of the 11th Annual International Symposium on Computer Architecture*, June 1984, pp. 340-347.

[Sequent 1988]
Sequent Computer Systems, Inc. Symmetry Technical Summary 1988.

[Thacker et al. 1988]
C. Thacker, L. Stewart, and E. Satterthwaite Jr. Firefly: a Multiprocessor Workstation. *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 909-920, Aug. 1988.

[Zahorjan et al. 1988]
J. Zahorjan, E. Lazowska, and D. Eager. Spinning Versus Blocking in Parallel Systems with Uncertainty. *Proceedings of the International Seminar on the Performance of Distributed and Parallel Systems*, North Holland, Dec. 1988.

The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems

John Zahorjan and Edward D. Lazowska
Department of Computer Science and Engineering
University of Washington

Derek L. Eager
Department of Computational Science
University of Saskatchewan

July 1989

Abstract

Spinning, or busy waiting, is commonly employed in parallel processors when threads of execution must wait for some event, such as synchronization with another thread. Because spinning is purely overhead, it is detrimental to both user response time and system throughput.

In this paper we study how spinning is affected by two environmental factors, multiprogramming and data-dependent execution times, and how the choice of scheduling discipline can be used to reduce the amount of spinning in each case. Both environmental factors increase the variation in delay until the awaited event occurs. In the case of multiprogramming, the thread that will eventually generate the event must compete with an uncertain number of other threads for processors. In the case of data-dependent behavior, the delay until the event occurs may depend strongly on the data presented at runtime.

We are interested in both the extent to which spin times increase over a simple baseline environment when multiprogramming or data-dependency is introduced and how this increase can be reduced through scheduling. The scheduling disciplines we examine are independent of the semantics of the parallel applications. They are thus applicable to the parallel solution of a wide variety of problems and to alternative approaches to solving any single problem independently of algorithm employed.

Our analysis is conducted using simple, abstract models of parallel hardware and software. We investigate two software models, one representing the archetypical construct of fork/join rendezvous and the other mutual exclusion using a lock. Our hardware model is most naturally applied to shared memory multiprocessors, since we assume that each thread is capable of running on many different processors during its execution with negligible penalty. Both simulation and exact analytic techniques are used to obtain performance results.

This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, Digital Equipment Corporation (the External Research Program and the Systems Research Center), and the Natural Sciences and Engineering Research Council of Canada. A portion of this work was done while Zahorjan was on sabbatical leave at Laboratoire MASI, University of Paris VI.

Authors' addresses: John Zahorjan and Edward D. Lazowska, Department of Computer Science and Engineering FR-35, University of Washington, Seattle, WA 98195; Derek L. Eager, Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada.

1. INTRODUCTION

When a thread of control on a parallel machine must wait for some event before proceeding, it may be reasonable for the thread to spin, that is, to sit in a tight loop continuously checking for the required condition. The time spent spinning is overhead, and thus diminishes the effective processing rate of the system.

While the primary factor ultimately limiting speedup in the solution of a particular problem is almost certainly lack of sufficient parallelism in the algorithm implemented, spin overhead can also have a significant effect [Gehringer et al. 1987, Zahorjan et al. 1988]. Further, unlike choice of algorithm, the amount of spinning can be affected by policies implemented in shared software such as the operating system or library routines. Thus, an understanding of how spinning is affected by alternative policy decisions is needed by software designers.

In this paper we consider how the amount of spinning exhibited by a parallel job is affected by two environmental influences, data-dependent behavior and multiprogramming, and how ill effects can be minimized by proper choice of software control policies. Because the policies we consider are generic, they are applicable to a wide variety of applications and even to different approaches to parallelizing the same application.

The dependence of thread times on input data is a natural characteristic of many applications. Because data dependence leads to unequal thread execution times, it makes balancing of work among threads difficult. This in turn leads to spinning as threads that finish "early" wait at synchronization points for the slower threads to catch up.

The motivation for multiprogramming in shared memory parallel computers is much like that in sequential systems: the desire to keep the processor(s) as usefully busy as possible. In a sequential system multiprogramming allows the execution of one job to occupy the processor during periods when another job is unable to make use of it, such as during I/O latency. In a parallel system, an individual job typically exhibits an amount of parallelism that varies over time. Thus, in general no single application can make constant use of all the processors in the system.

In estimating the effect of data-dependency and multiprogramming on spinning we begin by estimating the baseline spin overhead, defined to be that occurring in a uniprogrammed system for a job with easily predictable and homogeneous thread execution times. Because the system is uniprogrammed, the programmer knows in advance exactly how many physical processors will be available to his job, i.e., the total number, and in an ideal case can evenly partition the work of the job into a number of threads equal to the maximum physical parallelism. Thus, together our two assumptions of uniprogramming and predictable execution times represent in a rough sense the most favorable conditions under which spinning can be employed.

To reflect multiprogramming in our model, we assume that the number of processors allocated to a job can vary from run to run, and potentially even during a run. We assume that in response to this variability, each job splits into a number of threads equal to the total number of physical processors, thus enabling it to take advantage of any possible actual allocation of processors. On average, however, a single job will have fewer processors available to it than it has threads. Intuitively, we expect the reduced availability of physical processors to increase spin times.

To reflect data dependence, we smoothly increase the amount of variation among the execution times of different threads. As the variation increases, the potential maximal time spent spinning for an event also increases. Thus, in this case also intuitively we expect higher variation to lead to longer spin times.

It is clear that the amount of spinning realized by an application depends heavily on the precedence relations among its component threads. Further, it is reasonable to assume that the extent of any increase in spin overhead caused by environmental factors also depends on job structure. Because of this, we examine two different basic precedence structures that are exhibited in a large number of realistic parallel applications. The first involves spinning because of competition among threads. Here we assume a set of largely independent threads that use a lock to provide mutual exclusion when accessing some resource. When a thread wanting the resource finds the lock in use, it spins until the lock becomes free. The second structure involves spinning because of cooperation among threads. Here we model a set of threads that attempt to synchronize at a barrier. Each thread reaching the synchronization point (that is, the barrier) spins until all other threads have also reached the barrier.

The combination of the two workloads (locking and barrier synchronization) and three environments (baseline, multiprogramming, and data-dependency) involved in our study defines a set of six models. Each model is analyzed to obtain the amount of spin overhead that occurs. Some of the models admit exact analytical solutions. This is the analysis method of choice because it allows quick solutions given specific parameterizations of the model and because the analytic formulation of the solution can sometimes be used to illuminate qualitative aspects of the system, such as asymptotic behavior or optimal parameterizations. In some cases where it is not possible to obtain an analytical solution, an exact numerical procedure is given to compute the solution. This provides exact results, but only on a case by case basis. Finally, if neither exact approach can be applied, simulation is used to obtain estimates of performance.

Our results concerning how the environment affects the amount of spinning allow us to address the following questions:

For the multiprogramming environment:

- How does the performance of a system in which all processor allocation and thread dispatching decisions are made by the operating system compare with that of a system in which the operating system handles only the former while the application handles the latter?
- How can software strategies for thread dispatching be used to reduce the amount of spin overhead?
- How likely is it that *dynamic* processor allocation strategies, those that may change the allocation to an individual job during its lifetime, can be superior to *static* processor allocation policies?

For the data-dependency environment:

- For a fixed amount of data-dependence, how does the time required to achieve the barrier grow with the number of threads involved?
- For a fixed number of threads, how does barrier time grow with data-dependency?
- How can software strategies employed in the applications program be used to reduce the amount of spin overhead?

The work is also interesting because the models used make more realistic assumptions about the workload than most previous works, which have assumed some form of exponentially distributed times for all components of service.

In Section 2 we discuss in more detail the baseline models used in our study. Sections 3 and 4 present the multiprogrammed and data-dependent environments respectively. There we investigate the issues involved in defining appropriate models, and how the models are used to address the questions listed above. Section 5 contains our conclusions.

2. THE BASELINE MODELS

In this section we present the baseline models for the two workloads. As mentioned previously, the baseline models are useful principally for the purpose of comparison with the multiprogrammed and data-dependent environments considered subsequently.

2.1. Model Descriptions

The baseline model represents the ideal situation of a well-behaved job running in a controlled environment. In particular, we assume that there is only a single job in the system and that the execution time behaviors of its threads are very predictable.

The baseline models for both workloads represent the hardware as a set of P identical processors and the software (a single job) as a set of $T = P$ statistically identical threads. The two workload models differ in the behavior of the T threads.

For the lock contention workload, each job computes for an exponentially distributed amount of time with mean C and then attempts to capture the lock. The exponential distribution reflects variation in non-critical section work resulting from, for example, conditional execution outside of the critical section or non-symmetric thread functions.

When a computing thread requires the lock, if the lock is free the thread immediately captures it. Otherwise, the thread spins until the lock is released. If multiple threads are spinning when the lock is released, one of these threads is chosen at random to acquire the lock. The thread releasing the lock returns to the compute phase.

A thread obtaining the lock holds it for exactly L time units, and then releases it. This deterministic lock holding time reasonably approximates a large number of applications using locks. For example, a ubiquitous use of locks is to protect queues during enqueue and dequeue operations. These uses, and many others, are typified by very short lock holding times to perform a deterministic activity. Thus, lock holding times for an important class of applications are essentially deterministic.

We make the assumption that the effective computation rate of all threads is unaffected by the number of threads currently spinning. Anderson et al. [1989] have shown that naive implementations of spin locks can lead to considerable memory bandwidth contention in a bus-based system, but that more careful implementation of the lock mechanism can virtually eliminate this overhead. We expect the same sorts of behavior in systems with more complicated memory connection networks.

The baseline model of the barrier synchronization workload is quite similar to the lock contention model. Once again, there are P processors and $T = P$ threads. Each thread is in one of two states: computing or spinning. A thread computes for exactly C time units before reaching the barrier. If not all other threads have already reached the barrier (impossible for the baseline model but important in the two variations), it begins to spin. When the last thread reaches the barrier, all threads return to the computing state.

The deterministic compute times in the baseline model reflect the fact that barrier synchronization is most natural for software in which the thread execution times are very nearly equal. In the third variation of this model, data-dependency, we relax this assumption.

2.2. Solution of the Baseline Models

In this subsection we discuss the techniques used to analyze the models just described. In later subsections we discuss the performance measures obtained from the models and their implications.

For the barrier synchronization workload, our assumption of deterministic compute times yields a model whose analysis is trivial. In this idealistic case there is no spinning at all in the system and the barrier is achieved after C time units.

In contrast, the lock contention workload model is somewhat more complicated. Here our assumptions lead to the M/D/1/N model shown in Figure 1, where the infinite-server station represents the exponential compute time of each thread, the single FCFS server represents the critical section code executed while the lock is held, and the queue of that server represents spinning threads.

This M/D/1/N model was used previously to study lock contention by Dubois and Briggs [1982], who developed an heuristic solution technique for it. Their approach requires the solution of a Pth degree polynomial, and so involves an iteration. We have developed a new, non-iterative, approximate solution based on Mean Value Analysis [Reiser and Lavenberg 1980] that takes constant time independent of the model parameters [Zahorjan 1989]. Our initial investigations show this technique to be both faster and more reliable than the earlier approximation.

Both heuristic solution approaches gain execution speed by sacrificing some amount of accuracy. Thus, they are most appropriate when either a very large system is being studied or a large number of examples of smaller systems need to be analyzed. For this study, we have paid the execution time penalty of an *exact* analysis technique to obtain performance results for our examples. The approach we have used is applicable to systems with up to about 125 processors. This limitation in size is not imposed by computational costs so

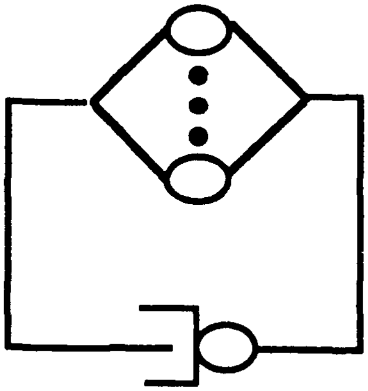


Figure 1 - The Baseline Lock Contention Model

much as numerical stability: the solutions can be very sensitive to transition probabilities that involve large powers of values near one and so are hard to calculate accurately.

The exact solution is found by examining the embedded Markov chain [Kleinrock 1975] defined by the service time completion instants. The state of this embedded chain represents the number of lock contenders that are left behind by a thread when it gives up the lock. Because we have deterministic lock holding times and exponential inter-lock request times, it is a simple matter to compute the distribution of the number of lock contenders that arrive during a lock service time, and so the transition rates among the states are easily found. These transition rates induce an upper triangular state transition matrix, whose solution can be found in time $O(T^2)$.

The solution of the embedded chain yields the probabilities $P_{departure}(n)$, $0 \le n \le T-1$, that n contenders remain just after a departure. These can be transformed into the equilibrium state probabilities $P_{equilibrium}(n)$, $0 \le n \le T-1$, in time $O(T)$ using the general relation [Dallery 1988]

$$P_{departure}(n) = \frac{\lambda(n)\, P_{equilibrium}(n)}{X}, \qquad 0 \le n \le T-1$$

where $\lambda(n)$ is the arrival rate of new lock requests when there are already n outstanding and X is the equilibrium lock throughput rate. (The relationship is obtained by observing that the departure and arrival instant distributions must be equal.) In our case, because inter-lock request times are exponential the arrival rates $\lambda(n)$ are equal to $\frac{T-n}{T}\frac{1}{C}$. The equilibrium distribution is completed using $P_{equilibrium}(T) = 1 - \sum_{n=0}^{T-1} P_{equilibrium}(n)$. From there, it is a simple matter to compute performance measures from the equilibrium state distribution.

2.3. Results for Lock Contention Workload

We let C be the unit of time, that is, C is set equal to 1.0. The two remaining parameters, P and L, are varied in all experiments to show their influence on performance.

Figures 2a and 2b show the amount of spinning that occurs in small (20 processor), medium (50 processor), and large (100 processor) baseline systems. The lock holding time L is chosen for each system size to achieve the utilizations listed on the X-axes. Thus, for a fixed lock utilization a larger system has a smaller lock holding time than a smaller system.

Figure 2a gives the mean number of processors spinning as a function of lock utilization. We note that the absolute number spinning for fixed lock utilization is nearly independent of system size.

Figure 2b gives the fraction of the total processing power of the system that is consumed by spinning on average, that is, each point of Figure 2a divided by the system size P. (This information can be interpreted equivalently as the fraction of time each processor spins.) Here we see that the effects of high lock utilization *diminish* with system size. For example, our largest system (100 processors) loses only 3% of its processing power to spinning for a 90% busy lock while our smallest system (20 processors) loses 10%. Thus, there is reason to expect that, for instance, a database system supporting a large number of users on a highly parallel

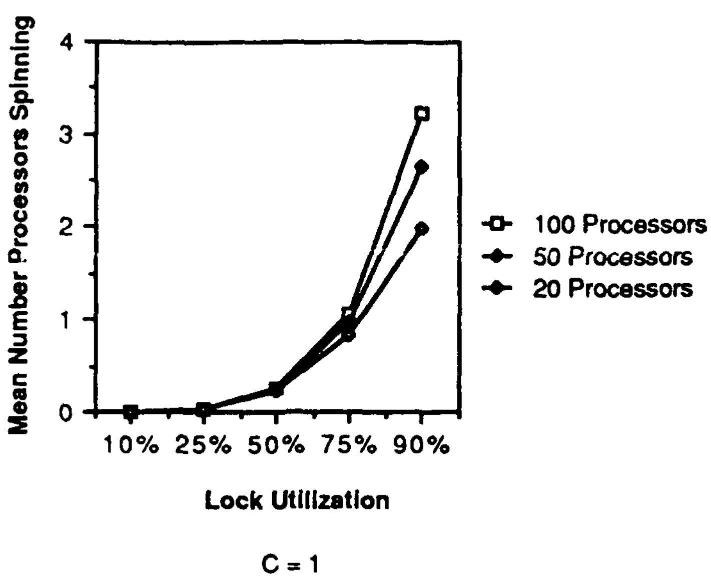


Figure 2a - Absolute Number Spinning

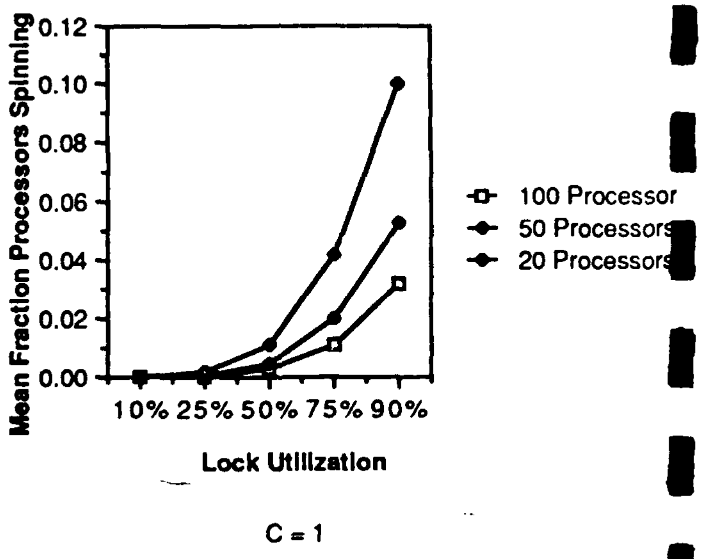


Figure 2b - Fraction Spinning

processor and using small locking granules (and so having small lock holding times) would be less sensitive to lock contention than a smaller system supporting fewer users and using coarser granules (and so longer lock holding times).

Perhaps more surprisingly, neither large nor small systems are severly handicapped by even a 90% busy lock. Thus, it would appear that a lock must be an extreme bottleneck before its effects on system performance become significant.

2.4. Results for the Barrier Synchronization Workload

Our baseline barrier synchronization workload model represents the ideal situation in which all computation can be parallelized and allocated perfectly evenly across any number of processors. As mentioned previously, this results in threads that naturally complete synchronously and so no spin overhead occurs. In subsequent sections we examine the extent to which compromising the ideal situation degrades performance.

3. THE MULTIPROGRAMMED SYSTEM

In the multiprogramming environment, we assume that any thread can run on any processor, and that thread execution time is unaffected by processor selection. This assumption ignores some possible second order effects, such as the cache related benefits of running a thread on the same processor each time it is dispatched, but is basically accurate for shared memory machines such as the Sequent [Lovett & Thakkar 1988] in which all memory is at a uniform distance from all processors.[1]

To model a multiprogramming system, we must represent the policy implemented by the scheduler. Choosing a scheduling policy is much more difficult in a parallel than a sequential system, and to date there is not much data on which approaches might be preferable. One basic distinction that can be made among these approaches is that between "thread based" and "job based" policies. In a thread based policy the scheduler does not use information about which job an individual thread belongs to in deciding when to schedule it. Thus, in the simplest case, the system maintains a single queue of all ready threads and serves them in round-robin fashion. This results in an allocation of processors to jobs that is proportional to the number of threads in each job. In contrast, job based schedulers use information about which job each thread belongs to. In the simplest case, there is a queue of ready threads for each job, and processors circulate in a round-robin fashion among the queues. Thus, each job receives the same rate of processor allocations independently of the number of threads it contains.

---

[1] Note that one can view thread service time in our model as including a component to represent the *average* overhead due to cache misses and invalidation traffic when a thread switches processors. A more detailed study to compare fixed with changing execution site strategies should be possible using the same basic structure as our model.

In this paper we are concerned only with the job based policies. (An earlier paper [Zahorjan et al. 1988] considers thread based scheduling.) Despite this restriction, there are still a considerable number of possible "basic approaches". Among these are:

Co-Scheduling [Ousterhout 1982]
The central idea is to schedule processors in a way that guarantees that all threads of a job are in service if any of them are. In the case of our model, where all jobs have exactly P threads, all P processors are swapped from one job to another at scheduling instants.

Static Allocation
A job receives some number $A \leq P$ of processors when it is initiated, and keeps that allocation throughout its lifetime. This approach is simple, and has been used in distributed environments [DeWitt et al. 1987] and proposed for use in parallel ones [Chen & Shin 1987].

Dynamic Allocation
The number of processors allocated to a job varies during the lifetime of that job.

We note that from the point of view of the implementation, the first two approaches are considerably simpler than dynamic allocation. This is true because (a) even given information on the non-spin resource usage of each thread, it might be hard for the operating system to decide when and how to reallocate processors, (b) obtaining the resource usage information is problematic since to the operating system a spinning thread and a computing thread look the same, and (c) some applications may be very sensitive to changes in their processor allocation, so that the cost of a poor reallocation decision could be high. Further, from the application's point of view there may not be much benefit to dynamic allocation, since it can be shown (under somewhat simplified conditions) that a static allocation equal to the average number of processors the application can use provides speedup no worse than one half that obtained if the application is always provided as many processors as it can currently make use of [Eager et al. 1989]. Thus, the static policies seem natural candidates for this initial study.

3.1. The Multiprogramming Models

As in the baseline model, our multiprogramming model considers explicitly only a single job of $T=P$ threads. The way in which we represent interference by other jobs for use of the P processors of the system depends on the scheduling policy considered.

For Co-Scheduling, we assume that all P processors are cycled from job to job in a round-robin fashion. Each job makes use of the processors for deterministic time Q, the quantum length. A cost S is incurred in switching the processors from one job to another. The effective total number of jobs in the system is a parameter varied in our experiments. Note that while Co-Scheduling is basically a processor allocation mechanism, under the assumptions of our model no thread dispatching policy is needed since there are never more threads than available processors while the job is running.

For all other scheduling policies examined, a single job is allocated a fixed number $A < P$ processors. The A processors are shared among the T threads of the job in a manner determined by the thread dispatching policy.

There are basically two approaches to thread dispatching. In the first, the operating system provides this function. This implies that the dispatching decisions are independent of the "state" of the threads (e.g. whether the thread is spinning or performing useful work), since in general this information is not available to the system. This approach is used in the Dynix [Lovett & Thakkar 1988] and Mach [Young et al. 1987] operating systems.

The alternative approach is to have the application itself perform thread dispatching. The advantage here is that thread state information is available and can be used in making decisions. This style of thread scheduling is exhibited by the Presto system [Bershad et al. 1988].

We examine two variations of operating system based thread dispatching. Under the "OS" policy, threads are served in a strict round-robin fashion, with all A processors being reallocated to new threads of the same job after each (deterministic) quantum of length Q. There is a cost S for performing the reallocation. The "Variable OS" policy is similar, but the length of a quantum is adjusted so that the amount of time each thread spends without a processor is held constant regardless of how few processors have been allocated or how many threads exist. Here, if W is the target amount of time a thread spends without a processor, we set Q equal to $W\frac{A}{T-A}$. (This is only an approximation in the case of our model because it ignores the context switch time S, but the exact expression to achieve an average of W is cumbersome and is closely approximated by the function above.) The Mach scheduler implements an adaptive Q policy of the same nature.

We also consider two variations of application-based thread scheduling. In the first, "Application", rotation of processors among threads is again based on quanta of length Q. However, when the quantum for a processor expires, the application's thread dispatcher is free to make any choice for the next thread to run, including the thread already running. (The cost to make this choice is always S independent of the decision.) In the case of our lock contention workload, the thread dispatcher never preempts a thread currently holding a lock. In the case of the barrier synchronization workload, those threads with the largest remaining service requirement are chosen for execution.

Under the last discipline that we consider, "Blocking", each thread runs until it reaches a synchronization point where it cannot immediately continue. At that time, and that time only, it releases the processor to a waiting thread. We consider this an application-based strategy because the decision whether to spin or block is typically made by the programmer.

3.2. *Solution of the Multiprogramming Models*

In our model, to obtain the performance of a Co-Scheduled system it suffices to obtain the performance of a uniprogramming system and then "factor in" the context switch overhead and contention for processors caused by other jobs. In particular, if a job requires time $Z_{uniprog}$ to complete in a uniprogrammed environment, in a Co-Scheduling environment with a total of n jobs in the system it requires time

$$Z_{co-sched}(n) = n\ Z_{uniprog}\frac{Q+S}{Q}$$

where Q is the quantum length and S is the context switch cost.

To model the other scheduling policies, we simply evaluate a model containing a single job of $T=P$ threads running on a fixed number $A<P$ of processors. Note that while this represents precisely the fixed allocation policies, it also represents in an approximate way systems employing dynamic policies. Here the fixed allocation in our model reflects the overall average effect of the changing allocation of the real system. Note also that in modelling the static policies there is no need to make assumptions about the composition of the multiprogramming mix (for example, we need not assume that all jobs in the system are identical) nor about the policy used for deciding how many processor to allocate to a job (since we examine the performance for $1 \le A \le P$).

For the lock contention workload, the multiprogramming models are too complicated to admit exact analysis. Further, they are sufficiently complicated that we have been unable to produce reliable analytic approximations. Thus, we have used simulation to obtain performance estimates.

We have run our simulations using standard confidence interval estimation techniques. In all cases we have run the simulations until the width of the 90% confidence intervals are no more than 5% of the point estimate for the mean value.

The analysis of the barrier synchronization workload model reduces to the analysis of the OS policy alone. The optimal application policy is to run those threads that have received the least service so far. This is equivalent to using the FCFS thread dispatching policy employed by the OS discipline. Thus the Application policy is identical to the OS policy. Similarly, the Blocking policy is identical to the OS policy when $Q=C$.

Thus, we can model the Blocking policy simply by this parameterization. We therefore restrict our attention to the analysis of the OS policy.

In contrast to the locking workload model, the barrier synchronization workload model admits an exact analytical solution. This is advantageous both because it allows solutions of specific examples to be obtained efficiently and more importantly because the equations describing the solution make clear how performance is affected by each model parameter.

The key to obtaining the solution is to observe that the barrier is achieved when the last thread obtains all C units of processor service it requires and that under reasonable assumptions about round-robin scheduling the last processor to complete service is the processor that is last to obtain its initial quantum of service.

Let the allocation of the A processors to a set of A threads during a quantum of length Q be called a round. Reaching the barrier requires (in general) a number of rounds. Without loss of generality, label the threads of the job so that threads 1 to A are those serviced in round 0 and the remaining threads are numbered successively according to their position in the round-robin queue. This situation is depicted in Figure 3 for the specific case of T=5, A=3, Q=1, and C=3.

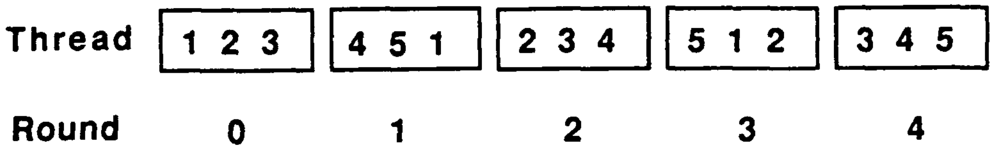


Figure 3 - Round Allocation for 5 threads on 3 processors

At the end of round 0, threads 1 through A will have acquired some amount of time $0 \le q < Q$ of computation towards the achievement of the current barrier. (This time q is accumulated at the end of the quantum during which the previous barrier was achieved.) No other thread will have made any progress toward the barrier. With these definitions, it is a simple matter to show that the following lemma holds:

Lemma:

If $Q < C$, the barrier is achieved when thread T completes its C units of computation. Otherwise, if $Q \ge C$, then if $q < C$ the barrier is achieved when thread A completes and if $q \ge C$ the barrier is achieved when thread T completes.

For simplicity of exposition, we consider only the case of $Q < C$ and $A < T$, so that thread T determines the completion time of the barrier. The modifications required for the remaining cases are easily obtained.

For $Q<C$ and $A<t$, it is clear that thread T completes during the $\left\lceil \frac{C}{Q} \right\rceil$th round in which it is allocated a processor, and that when it finishes during that round it will acquire time $q = Q\left\lceil \frac{C}{Q} \right\rceil - C$ time units towards the next barrier. Examining Figure 3, it is clear that thread T acquires its nth quantum of service during round $\left\lceil \frac{nT}{A} \right\rceil - 1$, and so must finish during round $Z = \left\lceil \frac{\left\lceil \frac{C}{Q} \right\rceil T}{A} \right\rceil - 1$. Thus, the barrier begins at time q before the end of round 0 and ends at time q before the end of round Z. The time to achieve the barrier, B, is therefore given by

$$B = q + \left[Z(Q+S) - q \right] = Z(Q+S)$$

where S is the context switch time required to reallocate the processors. From this we get that

$$\text{Average Number Spinning} = A\frac{Q}{Q+S} - \frac{TC}{B}$$

$$\text{Average Number Context Switching} = A\frac{S}{Q+S}$$

and

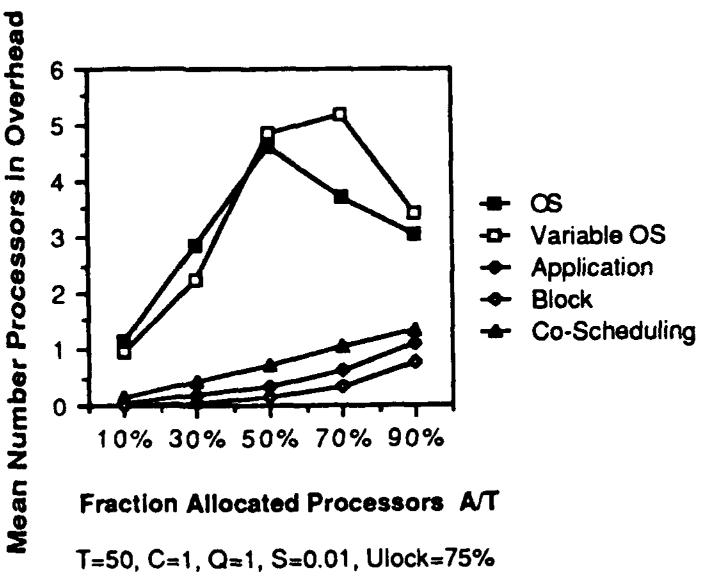
$$\text{Average Total Wasted} = A - \frac{TC}{B}$$

While these results have been obtained for the quantum-based policies, they can be applied to the Blocking policy in our model by observing that this is equivalent to the $Q=C$ quantum-based policy.

3.3. Results for Lock Contention Workload

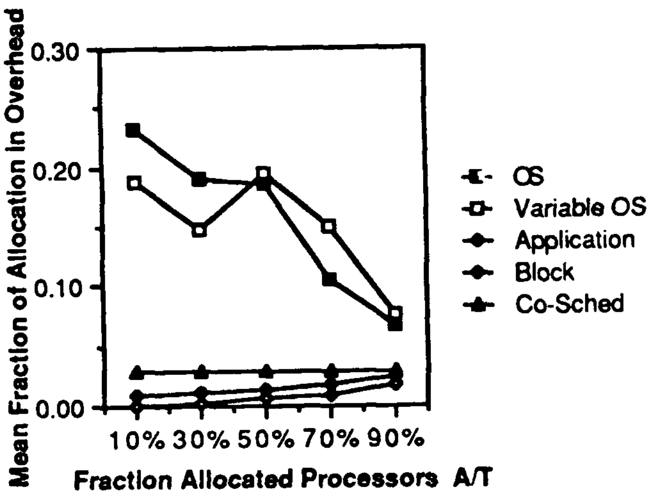
Figures 4a and 4b give basic results illustrating how the amount of spinning varies as a function of the number of processors allocated to a job and the job size. We illustrate specific results only for a job size of $T=50$ threads, but the results for other job sizes are similar. The scheduling quantum Q is set equal to 1.0, which is also the mean inter-lock request time C. The lock holding time L is chosen so that the job would result in 75% lock utilization if it were run on T processors. The scheduling overhead S is set to 0.01. (Experiments with the model indicate that performance is affected linearly over a large range for S near 0.01, but that its effect is small in all cases. This is examined in more detail in Figure 6.) The Variable OS results are computed by setting parameter W of that policy equal to 1.0, solving for the effective quantum length as a function of the number of processors allocated, and then analyzing the system using this modified Q.

The X-axis in Figure 4 represents A/T, that is, the ratio of the number of allocated processors to the number of threads. The Y-axis in Figure 4a gives the average number of processors that are consumed by overhead (the sum of spinning and context switching). For context switch times that are 1% of compute times, the vast bulk of this overhead (more than 80% typically) is due to spinning. The Y-axis in Figure 4b gives the fraction of the allocated processors consumed by overhead. If all jobs in the system were of the same type, this would also indicate the amount of overhead in the system as a whole.



T=50, C=1, Q=1, S=0.01, Ulock=75%

Figure 4a - Absolute Number in Overhead



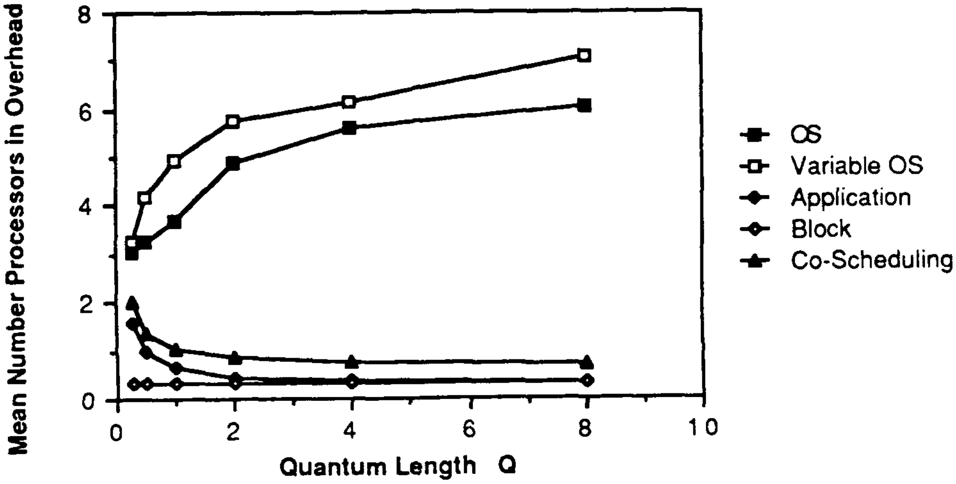
T=50, C=1, Q=1, S=0.01, Ulock=75%

Figure 4b - Fraction in Overhead

From Figure 4 it is clear that the two policies that preempt processors without regard to the state of the thread currently running (the OS and Variable OS policies) suffer a significant performance penalty. For our locking workload, this penalty results from the preemption of a thread holding the lock. While this happens only occasionally, it results in an avalanche of spinning. Because essentially all threads begin spinning before the lock can be released again, the high contention period for lock access lasts several scheduling quanta, and the probability that the lock is preempted remains abnormally high for quite some time.

Figure 4 leads us to conclude that the application should perform thread scheduling. However, another approach has been suggested. A spin lock technique that "nearly guarantees" that the thread holding the lock is not de-scheduled has been proposed for use in the NYU Ultracomputer [Edler et al. 1988]. However, because the mechanism is only approximate, some of the degradation illustrated in Figure 4 will still take place. Also, in general the condition under which one thread can inhibit the progress of others may be quite complicated, so that embedding this scheduling function in the locking mechanism is not of general use. Finally, Anderson et al. [1989] have shown that conventional spin lock techniques in bus-based parallel machines can lead to serious bus loading. The solution to this problem involves creating an ordered queue of threads waiting for the lock. While this approach solves the bus loading problem, it exacerbates the scheduling problem. With this locking mechanism the ill effects noted in Figure 4 arise if *any waiting thread* is preempted.

Figure 4 also indicates the Co-Scheduling performs somewhat worse than the application-based policies. The reason for this is that Co-Scheduling runs all competitors for the lock at once, thus maximizing contention for it. In contrast, the application-based approaches run only a subset of the competitors at any one time, thus reducing the amount of spinning.

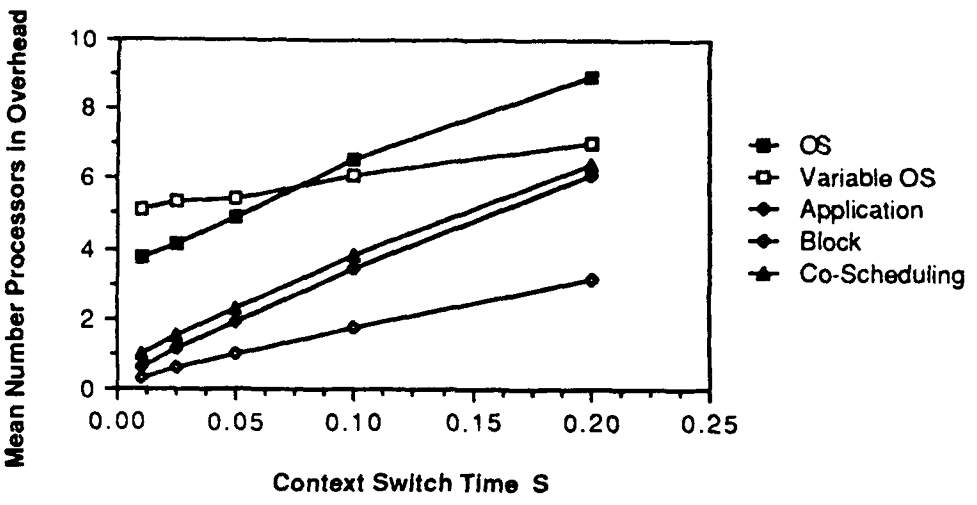


T=50, A=35, C=1, S=0.01, Ulock=75%

Figure 5 - Effect of Quantum Length

Figure 5 illustrates how performance is affected by the system choice of scheduling quantum, Q, for a job with 50 threads and a base lock utilization of 75%. The scheduling policies that may preempt a thread in the critical section behave very poorly as Q grows. In contrast, the other policies, which are relatively insensitive to Q, exhibit the opposite behavior due to the effects of context switch overhead for small Q. The relative insensitivity of these policies is highly desirable, since a real system will most likely exhibit a diversity of workloads making it difficult to tune parameters to which jobs are highly sensitive.

Figure 6 shows how performance is affected by context switch overhead S for a job with T=50 threads running on P=35 processors and parameterized so that lock utilization would be 75% if 50 processors were provided. (The quantum size Q is set to 1.0 for the quantum-based disciplines.) It is straightforward to interpret S as the efficiency with which a particular hardware and software system can perform this function. Perhaps a more interesting interpretation, however, is to consider the small S systems to represent coarse grained parallelism (parallelism at the level of individual procedures, for example) and the large S systems to represent fine grained parallelism (e.g., performing loop iterations in parallel). The effect of context switch overhead is nearly linear under all the policies. This is not surprising, since the context switch time under each policy does not affect the rate at which context switches take place, just their duration. The Variable OS policy shows the least sensitivity to context switch time because it has the longest effective quantum time for this parameterization of the model.

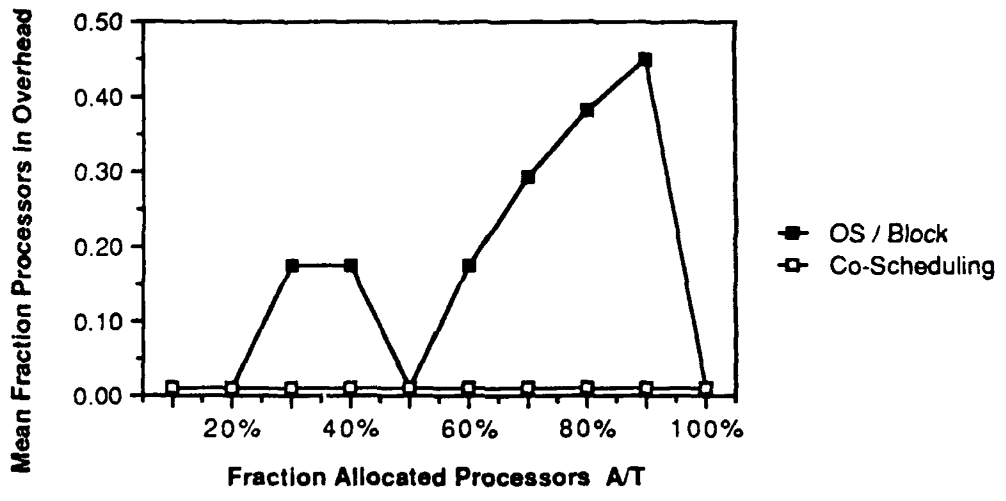


T=50, A=35, C=1, Q=1, Ulock=75%

Figure 6 - Effect of Context Switch Overhead

3.4. Results for Barrier Synchronization Workload

Figure 7 presents the basic effect of the processor allocation A on the performance of a barrier synchronization job. The results are stated as fractions of allocation consumed by overhead and apply to any system size T. The compute time C and the quantum length Q are set to 1.0, and the context switch overhead S is set to 0.01. Note that for this parameterization the Blocking and OS policies all equivalent. (No Application policy distinct from the OS policy exists.)

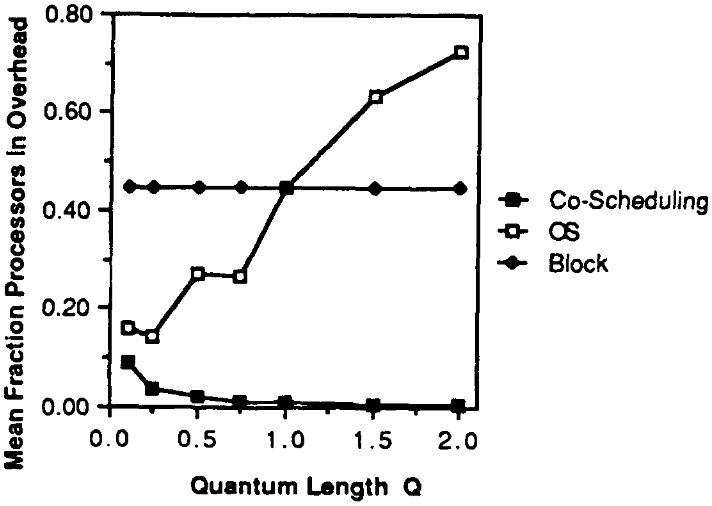


T=50, C=1, Q=1, S=0.01

Figure 7 - Effect of Allocation (Barrier Workload)

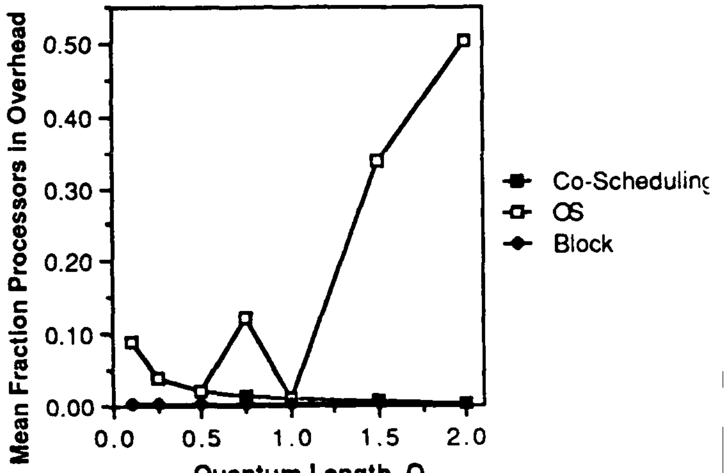
The striking feature of Figure 7 is the non-monotone behavior of overhead with allocation. This is explained by the nature of barrier synchronization. When the number of processors allocated divides equally the number of threads consumed by the barrier (and Q equals C), spinning can be avoided entirely. However, when the number of processors does not divide the number of threads, spinning must occur. Using the results of the previous subsection it is easy to show that the worst performance is obtained when the job is allocated $T-1$ processors, that is, one less than the number of threads it contains. For large system size T, fraction $1-\frac{1}{2(1+S)}$ of the processors are consumed by overhead under this allocation, or roughly half.

Figure 7 argues that a system that dynamically changes the number of processors allocated to jobs runs the risk of seriously degrading both system and user performance through the inappropriate usurping of even a single processor. It is natural in many circumstances for a parallel program to divide work into a number of pieces equal to the number of processors available to it. Once this is done, its response time and the efficient use of its processors depend on its processor allocation. A system wishing to change the allocation of a job, therefore, might be better off always halving the number of processors allocated rather than recapturing just one. By halving the number, jobs that have forked into a number of pieces working toward a barrier are guaranteed to always have an allocation that is a divisor of the number of branches in the fork, resulting in high processor efficiency.



T=50, A=45, C=1, S=0.01

Figure 8a - #Processors Doesn't Divide *T*



T=50, A=25, C=1, S=0.01

Figure 8b - #Processors Divides *T*

Figure 8 shows how the quantum length affects performance. Because the processor allocation has such a critical effect on performance, we show results both for an allocation that divides evenly the number of threads (e.g., 25 processors and 50 threads) and a poor allocation (in particular, 45 processors and 50 threads).

Because the quantum length in this case does not equal the compute time C, there are three distinct classes of scheduling disciplines to consider. One is the simple quantum-based policies, for example, the OS policy. These suffer under long quantum lengths because their only means to relinquish the processor is a quantum expiration. This causes completed threads to spin waiting for the barrier to be achieved while unfinished threads are waiting in the ready queue.

The Blocking discipline avoids the problem just mentioned, and for that reason seems like the natural policy to select when barrier synchronization is involved. However Figure 8 points out that in some circumstances the quantum-based policy can give better performance. This occurs when the processor allocation does not match the number of threads involved in the barrier. In this case using a small quantum length allows the processors to be shared roughly equally among all threads, so that they all finish at about the same time. Under the blocking policy, the threads originally allocated processors reach the barrier before any other threads begin computing. In the worst case, if there is only one more thread than allocated processors, all but one processor is wasted while this last thread is computing. Figures 7 and 8 together argue that a system that dynamically adjusts processor allocations might need to choose a small quantum to protect itself against poor allocation decisions.

The final policy, Co-Scheduling, seems to offer very good performance for this workload. (This is in fact the kind of workload for which Co-Scheduling was originally proposed.) In a real system, however, Co-Scheduling might perform much worse. The problem is that in general the number of threads involved in a barrier might be less than the number of processors in the system. In this case, during each quantum in which the machine is allocated to that job there are processors left idle. Ousterhout's original paper proposed a number of approaches to addressing this problem, however none of them appear capable of eliminating the effect. Thus, our model presents a perhaps unattainably optimistic bound on Co-Scheduling performance.

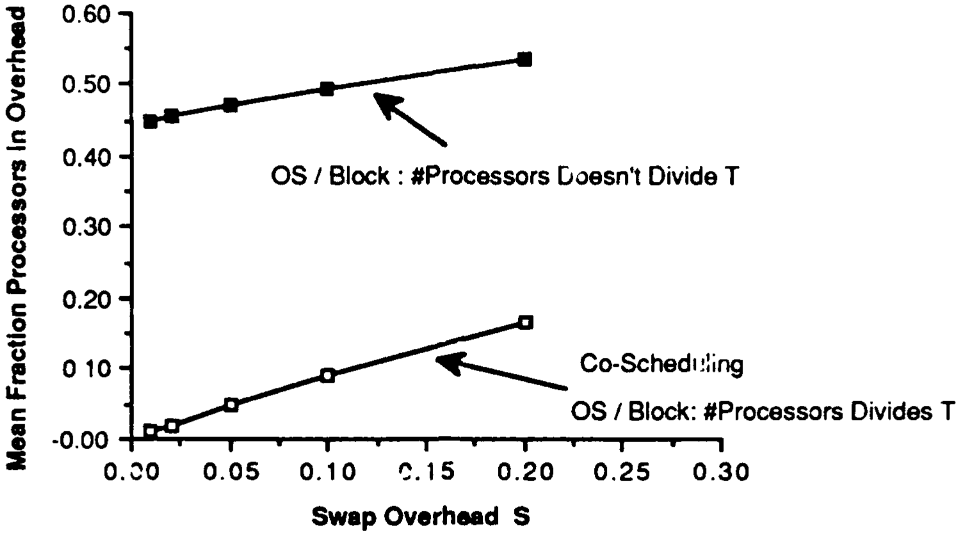


Figure 9 - Effect of Context Switch Overhead

Figure 9 shows how performance is affected by context switch time. Once again, the degradation is basically linear. In the quantum-based policies the swap overhead is given by $\frac{S}{Q+S}$ and in the Blocking policy by $\frac{S}{C+S}$.

4. THE DATA-DEPENDENT SYSTEM

To study the effect of data-dependence, we modify our baseline model in a different manner. As in the baseline model, we assume that the hardware is dedicated to a single job of $T=P$ threads. However, we relax the assumption of deterministic lock holding times (for the lock contention workload) or deterministic compute times (for the barrier synchronization workload). In particular, we allow these parameters to be uniformly distributed from $(1-f)M$ to $(1+f)M$, where M is the original deterministic time and f, $0 \le f \le 1$, is a parameter that reflects the amount of data-dependency. This method allows us to define a continuous spectrum of models where the service times vary from deterministic to randomly selected between zero and twice the mean.

There are a number of applications of our data-dependency model. For parallel systems supporting very large granularity, such as the Sequent, this variation in service times may reflect actual data-dependency. The assumptions made in our model apply most naturally to this type of system. Other interpretations of variation in execution time may be more natural for other kinds of systems. For systems supporting medium to small granularity, the service time variation reflects uncontrollable environmental influences on thread times such as page faults and cache hits in IO buffers. For fine granularity systems, such as those parallelizing individual loops, the variation reflects differences due to conditional code or inability to perfectly distribute the work [Polychronopoulos & Kuck 1987].

4.1. Solution of the Data-dependent Models

The data-dependent model for the locking workload is an M/G/1/N system. It is thus very similar to the model for the baseline case (shown in Figure 1). The solution technique used is identical to that case, i.e., the analysis of the embedded Markov chain given by task completion instants. The critical computation here is the distribution of the number of arrivals that occur during a lock holding time given that there are n threads in their computation phases at the beginning of the period. For a specific lock holding time of duration t, the probability that exactly d threads will finish during the holding time is

$$\binom{n}{d}(1-e^{-t/C})^d e^{-(n-d)t/C}$$

since each computing job makes a lock request after an independently selected exponential amount of time with mean C. The transition probabilities are now found by integrating the lock holding time t over the interval $((1-f)L,(1+f)L)$.

To find the average number of spinning processors for the barrier synchronization workload, we start with the observation that

$$\text{E[spinning]} = \frac{\int_{C(1-f)}^{C(1+f)} \text{E[amount of spinning} \mid \text{cycle length} = x\text{] Prob[cycle length} = x\text{]}\, dx}{\text{E[cycle length]}}$$

Now from

$$E\,\text{[amount of spinning} \mid \text{cycle length} = \text{x]} = \sum_{i=1}^{P-1} \int_{t=T(1-f)}^{x} (x-t)\text{Prob}[t \mid x]\,dt$$

where $\text{Prob}[t \mid x]$ is the probability density at t of the compute time of a thread that is not the slowest given that the slowest thread has compute time x, and from

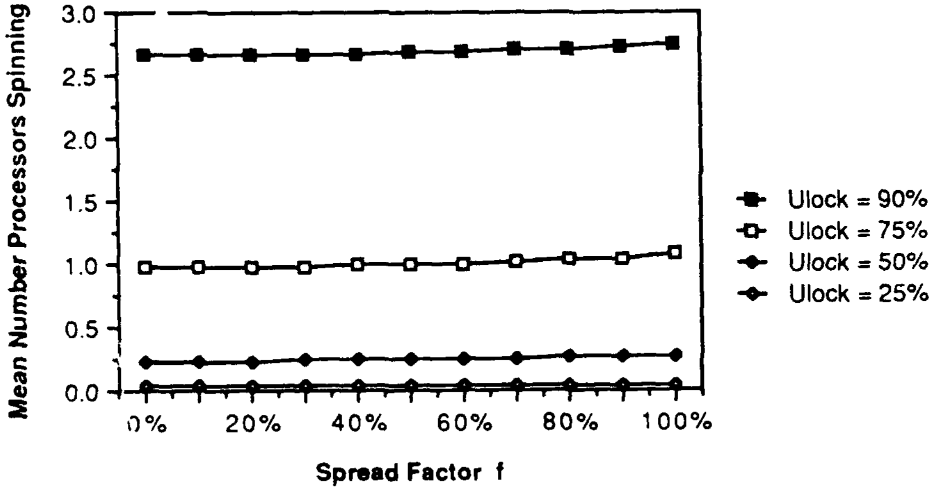
$$\text{Prob[cycle length} = \text{x]} = \frac{P}{2fT}\left[\frac{x-T(1-f)}{2fT}\right]^{P-1}$$

where Prob[cycle length = x] is the probability density of the cycle length distribution at x, it is possible to derive that

$$\text{E[number spinning processors]} = \frac{P(P-1)f}{(P+1)+(P-1)f}$$

4.2. Results for Data-dependent Models

Figure 10 shows the number of spinning processors as a function of spread f for the lock contention workload. Each line in the graph corresponds to different values of lock utilization obtained in the baseline (0% spread) case. We give results for the specific case of a job of 50 threads, but the results for other job sizes are very similar.



T=50, C=1

Figure 10 - Effect of Variability in Lock Holding Times

The clear conclusion to be drawn from these results is that variation in lock holding times of the magnitude we have considered is not a significant factor in the amount of spinning exhibited, regardless of the overall lock utilization. For a 90% busy lock, there is less than a 5% difference between the maximum amount of spinning ($f=1$) and the minimum ($f=0$). While the relative difference is greater for low lock utilizations (as much as 30% for a 10% busy lock), the absolute amount of spinning there is negligible in any case.

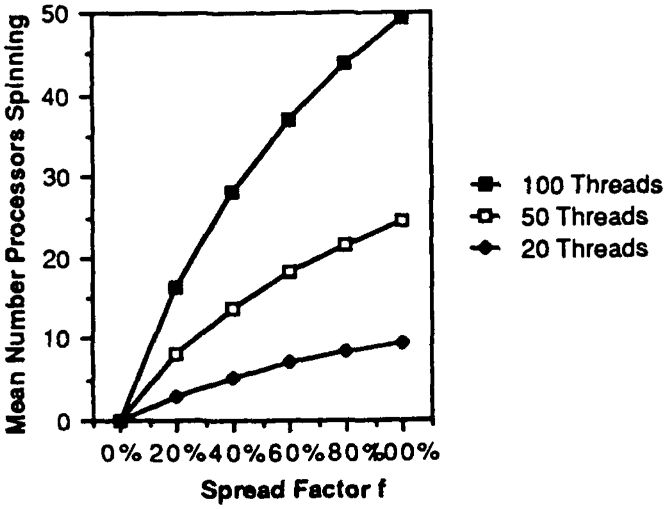


Figure 11a - Mean Number Spinning

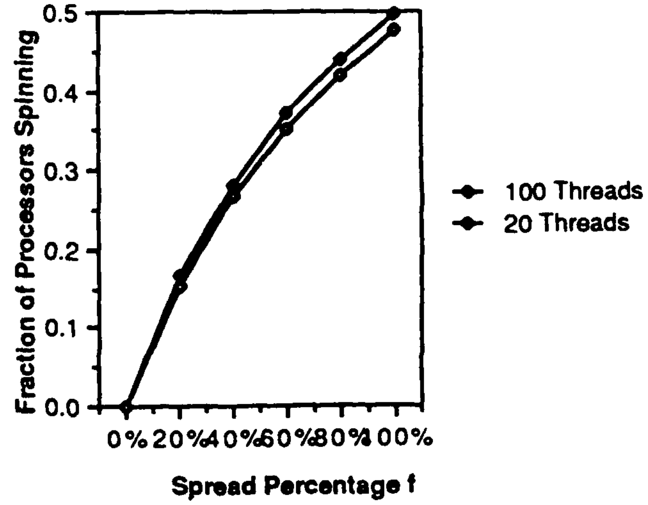


Figure 11b - Mean Fraction Spinning

The conclusion for the barrier synchronization workload is quite different, however. Figure 11a gives the amount of spinning as a function of the spread f for jobs of 20, 50, and 100 threads. As can be seen, the absolute amount of spinning increases quickly with f for all system sizes. Figure 11b gives the fraction of the total number of processors involved in spinning (or equivalently, the mean fraction of time each processor spends spinning) From this graph it is clear that the percentage degradation caused by variance in compute times is nearly independent of problem size.

Our analytic results for the barrier synchronization model provide further insight into this situation. Recall that the expected number of spinning processors is given by

$$\frac{P(P-1)f}{(P+1)+(P-1)f}$$

From this, it is easy to show that the maximum amount of spinning occurs at $f=1$, which agrees with our intuition. Here the expected number of spinning processors is $\frac{P-1}{2}$, or roughly 50% of the total number of processors for all but the smallest of systems.

For a fixed f we can examine the asymptotic behavior as the problem size grows, i.e., as $P \rightarrow \infty$. In this case, the total number of spinning processors approaches $\frac{Pf}{1+f}$, so that the fraction of the system involved in overhead goes to $\frac{f}{1+f}$. Figure 11b shows this asymptotic limit. Note that even a relatively small system of 20 threads closely approaches this worst case limit, and the 50 thread system has performance indistinguishable from it.

4.3. Using Finer Granularity to Reduce Spin Times for the Barrier Synchronization Workload

The results of the previous subsection indicate that even small variations in compute times can lead to large increases in spin times for the barrier synchronization workload. A natural question is what can be done to reduce this degradation in performance.

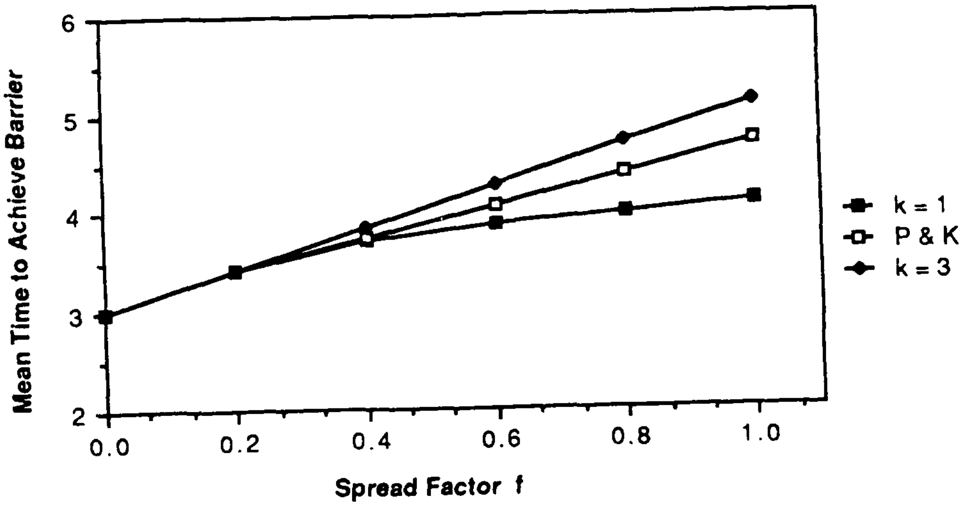
In the case that the T threads each represent inherently sequential work, it appears that nothing can be done to improve the situation. In this case, both the time to achieve the barrier and the total amount of computation to be performed are dictated by the problem, and so consequently is the amount of spinning.

On the other hand, imagine that the workload exhibits a great deal of parallelism, and that many independent tasks have been aggregated to form the T threads. This would be the case, for example, in parallelizing a FOR loop consisting of $L \equiv nP$ tasks (where each task is an independent iteration). For $n \leq 1$ this problem is identical to that just studied, and no reduction in spinning is possible. However, for $n > 1$ it might be possible to reduce the time required to reach the barrier through "self-scheduling" [Polychronopoulos & Kuck 1987]. In contrast to "static" scheduling policies that allocate all tasks to threads at the onset of the computation, self-scheduling policies delay (part of) the allocation decision to runtime. For example, each thread might initially be allocated one task, with the remaining tasks being put on a queue of work yet to be assigned. Each time a thread finishes its current task it takes the next task off the work queue. This dynamic allocation of work to threads helps reduce barrier times because all processors are kept busy as long as there are any unstarted tasks.

We study two kinds of self-scheduling policies, fixed [Kruskal & Weiss 1985] and variable [Polychronopoulos & Kuck 1987]. Under the fixed policies, each thread takes k tasks off the queue at a time, for some fixed k. (Note that the fixed self-scheduling policy with $k=n$ is equivalent to the static scheduling case studied in the previous section.) In general, smaller k provide better balancing of load (and so shorter barrier times) but also result in higher overheads due to the cost of operations on and contention for the work queue [Dritz & Boyle 1987].

The variable self-scheduling policy of Polychronopoulos and Kuck [1987] attempts to balance these two effects. Here the threads initially take large numbers of tasks each visit to the queue, but near the end of the computation take small numbers. In particular, their scheme requires each thread to take $\left\lceil \frac{r}{T} \right\rceil$ tasks, where r is the number of tasks currently on the work queue.

Figures 12 and 13 present the results of simulations that compare these two approaches to scheduling. We assume in all cases that each task has an execution time uniformly distributed between $C(1-f)$ and $C(1+f)$ (and that $C=1$). Further, we assume that operations on the work queue are infinitely fast. (We present results for non-negligible overhead subsequently.) These results, while optimistic of achievable performance, serve to highlight the ability of each discipline to balance load in the face of variation in task service times.

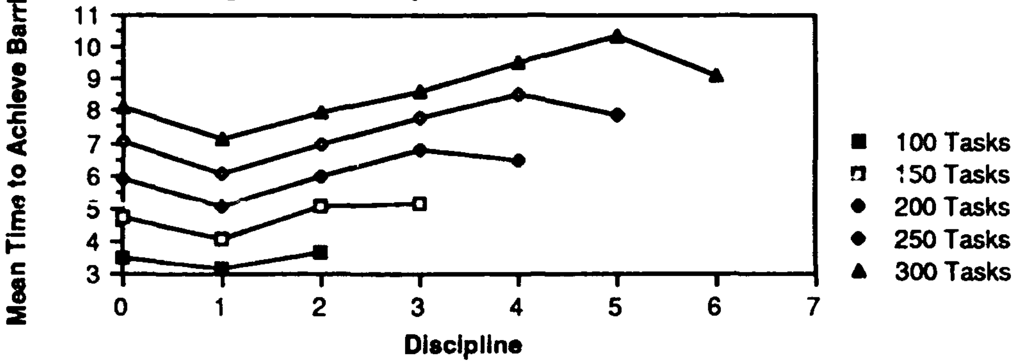


50 Threads, 150 Tasks, Task Time = 1.0

Figure 12 - Effect of Partitioning Scheme

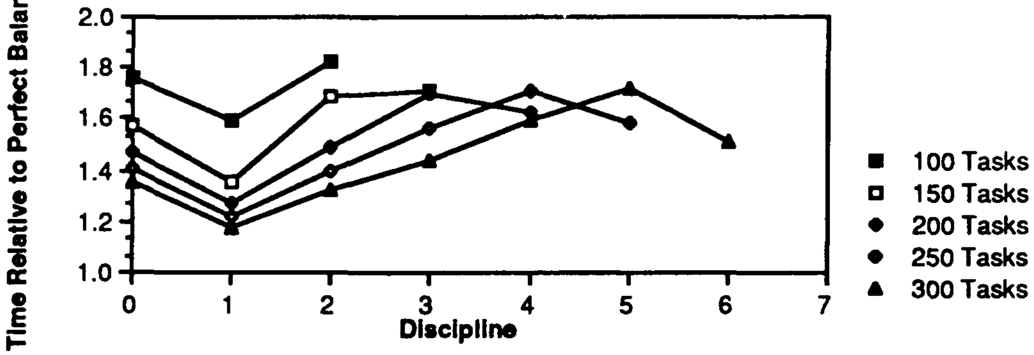
We consider first the robustness of the disciplines with respect to the variation in the task execution times, as reflected through the parameter f. Figure 12 compares three policies in a system with 50 threads executing a total of 150 tasks. The fixed policy with $k=3$ thus corresponds to pure static assignment: all tasks are

Figure 13a - Spread Factor f = 1.0



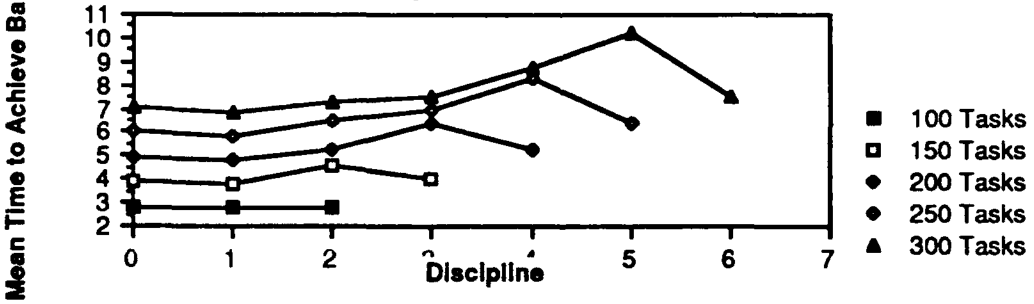
50 Threads, Task Time = 1.0

Figure 13b - Spread Factor f = 1.0



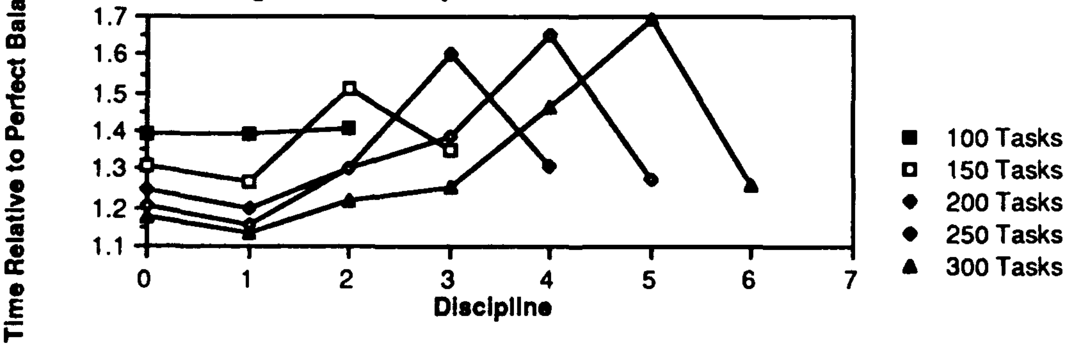
50 Threads, Task Time = 1.0

Figure 13c - Spread Factor f = 0.5



50 Threads, Task Time = 1.0

Figure 13d - Spread Factor f = 0.5



50 Threads, Task Time = 1.0

assigned before any of them have executed. In this sense, it is a worst-case approach with respect to load balancing. The $k=1$ policy on the other hand is the best-case with respect to load balancing. Finally, the variable policy (marked "P & K" in the figure) is a compromise between the two.

Figure 12 corroborates the evidence of the previous section: barrier times are very sensitive to variation in task execution times. However, note that the $k=1$ policy is much less sensitive than the other two. Further, it appears that the impact of variation "tops out" for this policy, while both the variable scheme and the purely static scheme show strongly rising barrier times at the limit of the variation possible under our model.

Figure 13 compares the static and self-scheduling policies for various problem sizes and amounts of variation in task service times. There are 50 threads available in all cases. Figures 13a and 13b correspond to the maximum variation case, $f = 1.0$. The former gives the absolute mean time required to achieve the barrier while the latter shows the ratio of the barrier time to that occurring under "perfect balancing", i.e., the total number of tasks divided by the number of threads (since each task has mean time 1.0). Figures 13c and 13d are the corresponding results for spread $f = 0.5$. In each case the X-axis gives the parameter k of the fixed self-scheduling policy indicating the number of tasks taken per visit to the work queue, except that at value 0 on the X-axis we plot the results for the variable self-scheduling policy of Polychronopoulos and Kuck. (For $k=n$ the fixed self-scheduling policy is equivalent to the purely static policy.)

We make a number of observations based on these results:

(a) *The variable allocation scheme works relatively well.*

In our examples the variable scheme achieves performance intermediate between that obtained for $k=1$ and $k=2$. Since we are ignoring work queue contention here, an actual system should show even better relative performance. In all cases we examined, the variable scheme achieved better performance than the fixed scheme requiring an equal number of visits to the work queue.

(b) *An improper choice of k for the fixed policies can greatly imbalance the load, independent of variation in task execution times.*

For example, in the case of $k=2$ with 150 tasks, the first 100 tasks are allocated equally among the threads, but the remaining 50 tasks can be executed by at most 25 threads. Thus, this choice results in worse performance than the purely static case, $k=3$, since the imbalance caused by an inappropriate group size exceeds that caused by variation in task workloads.

We note that this effect is the same as that discussed in Section 3.3 where we observed that in a multiprogramming system the number of processors allocated to a job should be a divisor of the number of threads involved in a barrier.

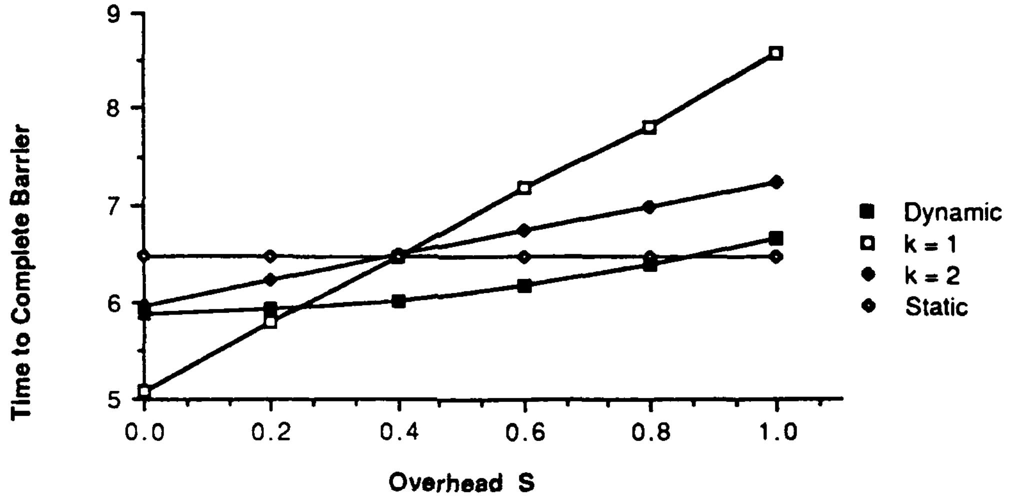
(c) *The effect of task service time variation decreases with increasing numbers of tasks.*

In graphs 13b and 13d, larger numbers of tasks per thread lead to performance closer to the best possible (modulo the "extraneous" results due to effect (b)). Further, the variation in performance due to choice of k declines with increasing number of tasks. Thus, factoring in work queue overheads, we expect to use larger k for larger problems.

This effect is easily explained by the central limit theorem, which says that the variation in the sum of n independent random variables decreases with increasing n. Thus, as the ratio of tasks to threads increases, the variation in individual task service times becomes less important.

Using a model similar to ours, Kruskal and Weiss [1985] concluded that the choice of k was not critical to performance. It appears from our figures that their conclusion was based at least partially on observation of only those cases for which n was substantial, thus muting the benefit of smaller k.

We end this section with a brief examination of the effect of overhead in accessing the work queue. This overhead has two components, the path length costs associated with obtaining work descriptors from the queue and possible wait time due to contention for access to a shared data structure. A number of researchers have investigated the performance of alternative approaches to maintaining the work queue [Dritz & Boyle 1987, Anderson et al. 1989]. Approaches applicable to large numbers of processors naturally require that the work queue be partitioned to allow concurrent access by more than one processor. We reflect these approaches in our model by allowing concurrent access to the work queue. While this is slightly optimistic (no approach can completely avoid competition while allowing access by any processor to any unfinished work), it is adequate for our purposes here. Finally, we note that the opposite extreme (assuming a naive implementation of the work queue that allows sequential access only) makes the comparison of the static and self-scheduling approaches very clear: self-scheduling is beneficial only if task granularity and variance in task service times are very large.



50 Threads, 200 Tasks, Task Time = 1.0

Figure 14 - Effect of Queue Access Overhead *S*

Figure 14 shows the performance of the static and self-scheduling policies for a system containing 50 threads and 200 tasks. The tasks have service times uniformly distributed between 0.0 and 2.0 (i.e., $f = 1.0$). We show the time required to achieve the barrier for overhead times from 0.0 to 1.0 (i.e., overhead times are given as fractions of the mean task completion time). We assume that the initial assignment of tasks to processors is performed without overhead, since this can be done statically. For instance, for the k=2 policy each thread is assigned its initial two tasks without overhead. A total of 50 overhead periods are subsequently required to dequeue the 100 tasks remaining after the initial assignment.

The main conclusion evident from Figure 14 is that the variable self-scheduling policy is quite robust with respect to overhead, in contrast to the fixed alternatives. The explanation for this lies in the fact that the variable policy behaves essentially as though it were a purely static policy applied to $\frac{T}{n}$ threads with n tasks assigned to each thread, where n equals the total number of tasks divided by T. In other words, the behavior of variable self-scheduling is controlled by the large initial assignments made to a fraction of its threads. For instance, in our example model with 50 threads and 200 tasks, thirteen threads are each given four tasks by the initial assignment and typically one of these groups of four tasks is the critical path in performing the barrier. This means that barrier completion time is controlled by a set of threads that do nearly no dynamic scheduling. This explains the insensitivity of the dynamic policy to overhead costs, as observed in Figure 14. This observation also explains the relatively high sensitivity of the dynamic policy to variation in task service time, as illustrated in Figure 12, as well as its performance relative to the k=1 and k=n policies when overhead is negligible (Figure 13).

5. CONCLUSIONS

We have examined how spin times in shared memory parallel systems are affected by two factors, multiprogramming and variability in thread execution times, using two canonical workloads. In both cases we have found that spin times, which represent overhead, can increase dramatically in some circumstances. We have investigated software strategies that may be implemented either in the operating system or by the user to reduce the sensitivity of the program to these effects.

In the specific case of multiprogramming, we have shown that decisions about how to allocate processors to jobs and how to schedule the threads of a job on its processors must be made cooperatively. This implies either the use of a global strategy that performs fixed allocation (i.e., does not change the allocation of processors to a job once the job has begun executing) or the use of a dynamic allocation strategy that involves a close alliance between the system's processor allocator and the application's thread scheduler. We have noted that in the range of workloads and systems considered, there appears to be very little difference in mean performance between synchronization via spinning and synchronization via blocking.

Our investigation of the effect of thread service time variability has demonstrated that the time required to achieve a barrier is extremely sensitive to even small (and tightly bounded) variations in individual thread completion times. In the case that the number of independent tasks to be performed exceeds the number of processors available, we have compared static scheduling of the tasks to both fixed and variable self-scheduling policies. We have found that the variable self-scheduling policy provides good overall performance and is by far the most robust with respect to overhead costs.

ACKNOWLEDGEMENTS

Partial support for this work was generously provided by Bell Communications Research, Boeing Computer Services, Digital Equipment Corporation, Tektronix, Inc., the Xerox Corporation, and the Weyerhauser Company. The Centre National de la Recherche Scientifique, France, and Laboratoire MASI, University of Paris VI, provided generous support and resources for Zahorjan for the year sabbatical leave during which part of this work was performed.

REFERENCES

[Anderson 1989]
Thomas E. Anderson. The Performance Implications of Lock Management Alternatives for Shared-Memory Multiprocessors. To appear, *Proc. 1989 International Conference on Parallel Processing.*

[Anderson et al. 1989]
Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. To appear, *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1989); forward by the program committee to *IEEE Trans. on Comp.*.

[Bershad et al. 1988]
Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice & Experience 18*,8 (August 1988), pp. 713-732.

[Chen & Shin 1987]
Processor Allocation in an N-Cube Multiprocessor Using Gray Codes, *IEEE Trans. on Comp. C-36*,12 (Dec. 1987).

[Dallery 1988] Yves Dallery, personal communication, Laboratoire MASI, Universite Pierre et Marie Curie, Paris, France.

[DeWitt et al. 1987]
David J. DeWitt, Raphael Finkel, and Marvin Solomon, The Crystal Multicomputer: Design and Implementation Experience, *IEEE Trans. on Soft. Eng. SE-13*, 8 (Aug. 1987).

[Dritz & Boyle 1987]
Kenneth W. Dritz and James M. Boyle. Beyond "Speedup": Performance Analysis of Parallel Programs. Technical Report ANL-87-7, Mathematics and Computer Science Division, Argonne National Laboratory, February 1987.

[Dubois & Briggs 1982]
M. Dubois and F.A. Briggs. An Approximate Analytical Model for Asynchronous Processes in Multiprocessors. *Proc. 1982 International Conference on Parallel Processing*, pp. 290-297.

[Eager et al. 1989]
Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Trans. on Comp. C-38*,3 (March 1989).

[Edler et al. 1988]
Jan Edler, Jim Lipkis, and Edith Schonberg. Process Management for Highly Parallel UNIX Systems, Ultracomputer Note #136, Courant Institute (April 1988).

[Gehringer et al. 1987]
Edward F. Gehringer, Daniel P. Siewiorek, and Zary Segall. *Parallel Processing: The Cm* Experience*, Digital Press (1987).

[Kleinrock 1975]
L. Kleinrock. *Queueing Systems: Volume I: Theory*. John Wiley and Sons, 1975.

[Kruskal & Weiss 1985]
Allocating Independent Subtasks on Parallel Processors, *IEEE Trans. on Soft. Eng. SE-11*, 10 (Oct. 1985).

[Lovett & Thakkar 1988]
The Symmetry Multiprocessor System, *Proc. 1988 Intl. Conf. on Par. Proc.* (Aug. 1988).

[Ousterhout 1982]
John K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proc. 3rd International Conference on Distributed Computing Systems* (October 1982), pp. 22-30.

[Polychronopoulos & Kuck 1987]
C.D. Polychronopoulos and D.J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. on Comp. C-36*,12 (December 1987), pp. 1425-1439.

[Reiser & Lavenberg 1980]
M. Reiser and S.S. Lavenberg. Mean Value Analysis of Closed Multichain Queueing Networks, *JACM 27*, 2 (April 1980).

[Young et al. 1987]
Michael Young et al. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, *Proc. 11th ACM Symp. on Op. Sys. Prin.* (Nov. 1987).

[Zahorjan et al. 1988]
John Zahorjan, Edward D. Lazowska, and Derek L. Eager. Spinning Versus Blocking in Parallel Systems with Uncertainty. *Proc. International Symposium on Performance of Distributed and Parallel Systems*, December 1988.

[Zahorjan 1989]
The Efficient Computation of Lock Contention Times in Parallel Systems, In preparation.

Thread Management for Shared-Memory Multiprocessors

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy

Department of Computer Science and Engineering
University of Washington
Seattle WA 98195

September 30, 1989

Abstract

Threads, or "lightweight processes," have become a common and necessary component of new languages and operating systems. Threads allow the programmer or compiler to express, create, and control parallel activities, contributing to the structure and performance of programs.

In this paper, we discuss the many alternatives that present themselves when designing a support system for threads on a shared-memory multiprocessor. These alternatives influence the ease, granularity, and performance of parallel programming. We conclude with a brief survey of three contemporary thread management systems (Mach, Presto, and Multilisp), using them to illustrate the issues raised in this paper.

Index Terms – thread, multiprocessor, operating system, parallel programming, performance

---

This material is based on work supported by the National Science Foundation (Grants CCR-8619663, CCR-8700106, CCR-8703049, and CCR-8907666), the Naval Ocean Systems Center, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program). Anderson was supported by an IBM Graduate Fellowship, and Bershad by an AT&T Ph.D. Scholarship.

1 Introduction

Disciplined concurrent programming can improve the structure and performance of computer programs on both uniprocessor and multiprocessor systems. As a result, support for *threads*, or "lightweight processes," has become a common element of new operating systems and programming languages.

A thread is a sequential stream of instruction execution. A thread differs from the more traditional notion of a "heavyweight process" in that it separates the notion of execution from the other state needed to run a program (e.g., an address space). A single thread executes a portion of a program, while cooperating with other threads that are concurrently executing the same program. Much of what is normally kept on a per-heavyweight-process basis can be maintained in common for all threads in a single program, yielding dramatic reductions in the overhead and complexity of a concurrent program.

Concurrent programming has a long history. The operation of programs that must handle real-world concurrency (e.g., operating systems, database systems, and network file servers) can be complex and difficult to understand. Dijkstra [Dijkstra 68] and Hoare [Hoare 74, Hoare 78] showed that these programs can be simplified when structured as cooperating sequential threads that communicate at discrete points within the program.

Multiprocessors offer an opportunity to use concurrency in parallel programs to improve performance, as well as structure. Moderately increasing a uniprocessor's power can require substantial additional design effort, as well as faster and more expensive hardware components. But, once a mechanism for interprocessor communication has been added to a uniprocessor design, the system's peak processing power can be increased by simply adding more processors. A shared-memory multiprocessor is one such design in which processors are connected by a bus to a common memory.

Multiprocessors lose their advantage if this processing power is not effectively utilized. If there are enough independent sequential jobs to keep all of the processors busy, then the potential of a multiprocessor can be easily realized: each job can be placed on a separate processor. However, if there are fewer jobs than processors, or if the goal is to execute single applications more quickly, then the machine's potential can only be achieved if individual programs can be parallelized in a cost-effective manner. Three factors contribute to the cost of using parallelism in a program:

Thread Overhead: The work, in terms of processor cycles, required to create and control a thread must be appreciably less than the work performed by that thread on behalf of the program. Otherwise, it is more efficient to do the work sequentially, rather than use a separate thread on another processor.

Communication Overhead: Again in terms of processor cycles, the cost of sharing information between threads must be less than the cost of simply computing the information in the context of each thread.

Programming Overhead: A less tangible metric than the previous two, programming overhead reflects the amount of human effort required to construct an efficient parallel program.

High overhead in any of these areas makes it hard to build efficient parallel programs. Costly threads can only be used infrequently. Similarly, if arranging communication between threads is slow, then the application must be structured so that little inter-thread communication is required. Finally, if managing parallelism is tedious or difficult, then the programmer may find it wise to sacrifice some speedup for a simpler implementation. Few algorithms parallelize well when constrained by high thread, communication, and programming costs, although many can flourish when these costs are low.

Low overhead in these three areas is the responsibility of the thread management system, which bridges the gap between the physical processors (the suppliers of parallelism) and an application (its consumer). In this paper, we discuss the issues that arise in designing a thread management system to support low-overhead parallel programming for shared-memory multiprocessors. In the next section, we describe the functionality found in thread management systems. Section 3 discusses a number of thread design issues. In Section 4, we survey three systems for shared-memory multiprocessors, Mach [Tevanian et al. 87], Presto [Bershad et al. 88], and Multilisp [Halstead 85], focusing our attention on how they have addressed the issues raised in this paper.

2 Thread Management Concepts

2.1 Address Spaces, Threads, and Multiprocessing

An address space is the set of memory locations that can be generated and accessed directly by a program. Address space limitations are enforced in hardware to prevent incorrect or malicious programs in one address space from corrupting data structures in others. Threads provide concurrency within a program, while address spaces provide failure isolation between programs. These are orthogonal concepts, but the interaction between thread management and address space management defines the extent to which data sharing and multiprocessing are supported.

The simplest operating systems, generally those for older-style personal computers, support only a single thread and a single address space per machine. A single address space is simpler and faster since it allows all data in memory to be accessed uniformly. Separate address spaces are not needed on dedicated systems to protect against malicious users; software errors can crash the system but at least are localized to one user, one machine.

Even single-user systems can have concurrency, however. More sophisticated systems, such as Xerox's Pilot [Redell et al. 80], provide only one address space per machine, but support multiple threads within that single address space. Because any thread can access any memory location, Pilot provides a compiler with strong type-checking to decrease the likelihood that one thread will corrupt the data structures of another.

Other operating systems, such as UNIX, provide support for multiple address spaces per machine, but only one thread per address space. The combination of a UNIX address space with one thread is called a UNIX *process*; a process is used to execute a program. Since each process is restricted from accessing data that belongs to other processes, many different programs can run at the same time on one machine, with errors confined to the address space in which they occur. Processes are able to cooperate by sending messages back and forth via the operating system. Passing data through the operating system is slow, however; only parallel programs that require infrequent communication can be written using threads in disjoint address spaces.

Instead of using messages to share data, processes running on a shared-memory multiprocessor can communicate directly through the shared memory. Some UNIX systems allow memory regions to be set up as shared between processes; any data in the shared region can be accessed by more than one process without having to send a message by way of the operating system. The Sequent Symmetry's DYNIX [Sequent 88] and Encore's UMAX [Encore 86] are operating systems that provide support for multiprocessing based on shared memory between UNIX processes.

More sophisticated operating systems for shared-memory multiprocessors, such as CMU's Mach and DEC SRC's Taos [Thacker et al. 88], support multiple address spaces *and* multiple threads within each address space. Threads in the same address space communicate directly with one another using shared memory; threads communicate across address space boundaries using messages. The cost of creating new threads is significantly less than that of creating whole address spaces, since threads in the same address space can share per-program resources. Figure 1 illustrates the various ways in which threads and address spaces can be organized by an operating system.

2.2 Basic Thread Functionality

At its most basic level, a thread consists of a program counter (PC), a set of registers, and a stack of procedure activation records containing variables local to each procedure. A thread also needs a control block to hold state information used by the thread management system: a thread can be *running* on a processor, *ready-to-run* but waiting for a processor to become available, *blocked*

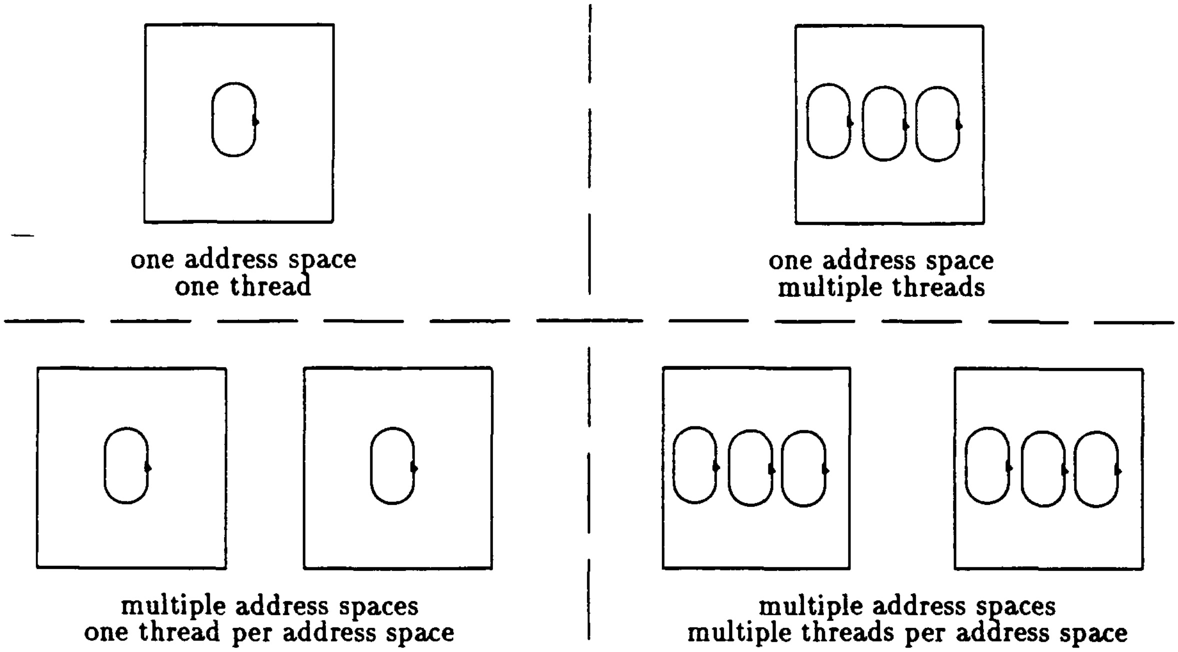

one address space
one thread
one address space
multiple threads
multiple address spaces
one thread per address space
multiple address spaces
multiple threads per address space

Figure 1: Threads and address spaces

waiting for some other thread to communicate with it, or *finished.* Threads that are ready-to-run are kept on a *ready-list* until they are picked up by an idle processor for execution. There are four basic thread operations:

Spawn: A thread can create or "spawn" another thread, providing a procedure and arguments to be run in the context of a new thread. The spawning thread allocates and initializes the new thread's control block and places the thread on the ready-list.

Block: When a thread needs to wait for an event, it may block (saving its PC and registers) and relinquish its processor to run another thread.

Unblock: Eventually, the event for which a blocked thread is waiting occurs. The blocked thread is marked as ready-to-run and placed back on the ready-list.

Finish: When a thread completes (usually by returning from its initial procedure), its control block and stack are deallocated, and its processor becomes available to run another thread.

When threads can communicate with one another through shared memory, *synchronization* is necessary to ensure that threads don't interfere with each other and corrupt common data structures. For example, if two threads each try to add an element to a doubly-linked list at the same time, one or the other element may be lost, or the list could be left in an inconsistent state.

Locks can solve this problem by providing mutually exclusive access to a data structure or region of code. A lock is acquired by a thread before it accesses a shared data structure; if the lock is held by another thread, the requesting thread blocks until the lock is released. (The code that a thread executes while holding a lock is called a *critical section.*) By serializing accesses, the programmer can ensure that threads only see and modify a data structure when it is in a consistent state.

When a program's work is split among multiple threads, one thread may store a result read by another thread. For correctness, the reading thread must block until the result has been written. This data dependency is an example of a more general synchronization object, the *condition variable*, which allows a thread to block until an arbitrary condition has been satisfied. The thread that makes the condition true is responsible for unblocking the waiting thread.

One special form of a condition variable is a *barrier*, which is used to synchronize a set of threads at a specific point in the program. In the case of a barrier, the arbitrary condition is "have all threads reached the barrier?" If not, a thread blocks when it reaches the barrier. When the final thread reaches the barrier, it satisfies the condition and *raises* the barrier, unblocking the other threads.

If a thread needs to compute the result of a procedure in parallel, it can first spawn a thread to execute the procedure. Later, when the result is needed, the thread can perform a *join to wait* for the procedure to finish and return its result. In this case, the condition is "has a given thread finished?" This technique is useful for increasing parallelism, since the synchronization between the caller and the callee takes place when the procedure's result is needed, rather than when the procedure is called.

Locks, barriers and condition variables can all be built using the basic block and unblock operations. Alternatively, a thread can choose to *spin-wait* by repeatedly polling until an anticipated event occurs, rather than relinquishing the processor to another thread by blocking. Although spin-waiting wastes processor time, it can be an important performance optimization when the expected waiting time is less then the time it takes to block and unblock a thread. For example, spin-waiting is useful for guarding critical sections that contain only a few instructions.

2.3 Why Not UNIX Processes?

It might seem that UNIX processes could be used directly as threads for parallel programming. Processes, like threads, can be spawned, blocked, unblocked and finished (in UNIX terms, `fork`, `pause`, `kill`, and `exit`); given shared memory, processes can inexpensively share data with each other.

Unfortunately, high overheads make the UNIX process interface ill-suited for many kinds of parallel programming. Creating a new process is expensive since it entails initializing and maintaining a great deal of state information. For instance, page tables, swap images and file descriptors are all kept on a per-process basis. Threads can be more efficient to create and manage than processes since they share this state. Further, controlling a UNIX process, once started, is slow and cumbersome because the operating system interface and implementation were not originally intended to support multiprocessing.

The difference in cost between UNIX processes and threads can be several orders of magnitude. On the Sequent Symmetry, a shared-memory multiprocessor, it takes about twenty milliseconds to spawn and terminate a "null" process. On the same hardware using a carefully implemented thread management system, a "null" thread can be spawned and terminated in as few as thirty *microseconds*, within an order of magnitude of the time to do a procedure call. Further, a context-switch between processes on the Sequent takes about six hundred microseconds, whereas a thread context-switch takes about twenty. The high cost of UNIX processes prevents them from being used to reflect the inherent structure of the concurrency in many applications.

3 Issues In Thread Management

This section considers the issues that arise in designing and implementing a thread management system as they relate to the programmer, the operating system, and the performance of parallel programs.

3.1 Programmer Issues

3.1.1 Programming Models

The flexibility to adapt to different programming models is an important attribute of thread systems. Parallelism can be expressed in many ways, each requiring a different interface to the thread system and making different demands on the performance of the underlying implementation. At the same time, a thread system that strives for generality in handling multiple models is likely to be well-suited to none.

One general principle is that the programmer should choose the most restrictive form of synchronization that provides acceptable performance for the problem at hand. For coordinating access to shared data, messages are a more restrictive, and for many kinds of parallel programs, more appropriate form of synchronization than locks and condition variables. Threads share information by explicitly sending and receiving messages to one another, as if they were in separate address spaces, except that the thread system uses shared memory to efficiently implement message-passing.

There are some cases where explicit control of concurrency may not be necessary for good parallel performance. For instance, some programs can be structured around a Single Instruction Multiple Data (SIMD) model of parallelism. With SIMD, each processor executes the same instruction in lockstep, but on different data locations. Because there is only one program counter, the programmer need not explicitly synchronize the activity of different processors on shared data, thus eliminating a major source of confusion and errors.

Perhaps the simplest programmer interface to the thread system is none at all: the compiler is completely responsible for detecting and exploiting parallelism in the application. The programmer can then write in a sequential language; the compiler will make the transformation into a parallel program. Nevertheless, the compiled program must still use some kind of underlying thread system, even if the programmer does not. Of course, there are many kinds of parallelism that are difficult for a compiler to detect, so automatic transformation has a limited range of use.

3.1.2 Language Support

Threads can be integrated into a programming language; they can exist outside the language as a set of subroutines that explicitly manage parallelism; or they can exist both within and outside the language, with the compiler and programmer managing threads together.

Language support for threads is like language support for object-oriented programming or garbage collection — it can be a mixed blessing. On one hand, the compiler can be made responsible for common bookkeeping operations, reducing programming errors. For example, locks can automatically be acquired and released when passing through critical sections. Further, the types of the arguments passed to a spawned procedure can be checked against the expected types for that procedure. This is difficult to do without compiler support.

On the other hand, language support for threads increases the complexity of the compiler, an important factor if a multiprocessor is to support more than one programming language. Further, the concurrency abstractions provided by a single parallel programming language may not do quite what the programmer wants or needs, making it necessary to express solutions in ways that are unnatural or inefficient.

A reasonable way of getting most of the benefits of language support without many of the disadvantages is to define both a language and a procedural interface to the thread management system. Common operations can be handled transparently by the compiler, but the programmer can directly call the basic thread management routines when the standard language support proves insufficient.

3.1.3 Granularity of Concurrency

The frequency with which a parallel program invokes thread management operations determines its *granularity*. A *fine-grained* parallel program creates a large number of threads, or uses threads that frequently block and unblock, or both. Thread management cost is the major obstacle to fine-grained parallelism. For a parallel program to be efficient, the ratio of thread management overhead to useful computation must be small. If thread management is expensive, then only *coarse-grained* parallelism can be exploited.

More efficient threads allow programs to be finer-grained, which benefits both structure and performance. First, a program can be written to match the structure of the problem at hand, rather than the performance characteristics of the hardware on which the problem is being solved. Just as a single-threaded environment on a uniprocessor can prevent the programmer from composing a program to reflect the problem's logical concurrency, a coarse-grained environment can be similarly restrictive. For example, in a parallel discrete-event simulation, physical objects in the simulated system are most naturally represented by threads that simulate physical interactions by sending messages back and forth to one another; this representation is not feasible if thread operations are too expensive.

Performance is the other advantage of fine-grained parallelism. In general, the greater the length of the ready-list, the more likely it is that a parallel program will be able to keep all of the available processors busy. When a thread blocks, its processor can immediately run another thread provided one is on the ready-list. With few threads though, as in a coarse-grained program, processors idle while threads do I/O or synchronize with one another.

The performance of a fine-grained parallel program is less sensitive to changes in the number of processors available to an application. For example, consider one phase of a coarse-grained parallel program that does fifty CPU-minutes worth of work. If the program creates five threads on a five processor machine, the phase finishes in just ten minutes. But, if the program runs with only four processors, then the execution time of the phase *doubles* to twenty minutes: ten minutes with four processors active followed by ten minutes with one processor active. (Preemptive scheduling, which could be used to address this problem, has a number of serious drawbacks, which are discussed in Section 3.2.2.) If the program had originally been written to use fifty threads, rather than five, then the phase could have finished in only thirteen minutes — a reasonable degradation in performance.

Of course, one could argue that the programmer erred in writing a program that was dependent on having exactly five processors. The program should have been parameterized by the number of processors available when it starts. But, even so, good performance can't be ensured if that number can vary, as it can on a multiprogrammed multiprocessor. We consider further the issues of multiprogramming in the next section.

3.2 Operating System Issues

3.2.1 Multiprogramming

Multiprogramming on a uniprocessor improves system performance by taking advantage of the natural concurrency between computation and I/O. While one program waits for an I/O request, the processor can be running some other program. Because the processor and I/O devices are kept busy simultaneously, more jobs can be completed per unit time than if the system ran only one program at a time.

A multiprogrammed multiprocessor has an analogous advantage. Ideally, periods of low parallelism in one job can be overlapped with periods of high parallelism in another job. Further, multiprogramming allows the power of a multiprocessor to be used by a collection of simultaneously running jobs, none of which by itself has enough parallelism to fully utilize the multiprocessor.

3.2.2 Processor Scheduling

Processor scheduling can be characterized by whether physical processors are assigned directly to threads or are first assigned to jobs and then to threads within those jobs. The first approach, called *one-level* scheduling, makes no distinction between threads in the same job and threads in different jobs. Processors are shared across all runnable threads on the system so that all threads make progress at relatively the same rate. In this case, threads from all jobs are placed on one ready-list that supplies all processors, as shown in Figure 2. Although this scheme makes sense for a uniprocessor operating system, it has some unpleasant performance implications on a multiprocessor.

The most serious problem with one-level scheduling occurs when the number of runnable threads exceeds the number of physical processors, because preemptive scheduling is necessary to allocate processor time to threads in a fair manner. With preemption, a processor can be taken away from one thread and given to another at any time. In a sequential program, preemption has a well-defined effect: the program goes from the running state to the not-running state as its one thread is preempted. The effect of preemption on the performance of a sequential program is also well-defined: if n CPU-intensive jobs are sharing one processor in a preemptive, round-robin fashion, then each job receives $1/n$th the processor and is slowed down by a factor of n (modulo the preemption and scheduling overhead).

For a parallel program, though, the effects of "untimely" processor preemption on performance

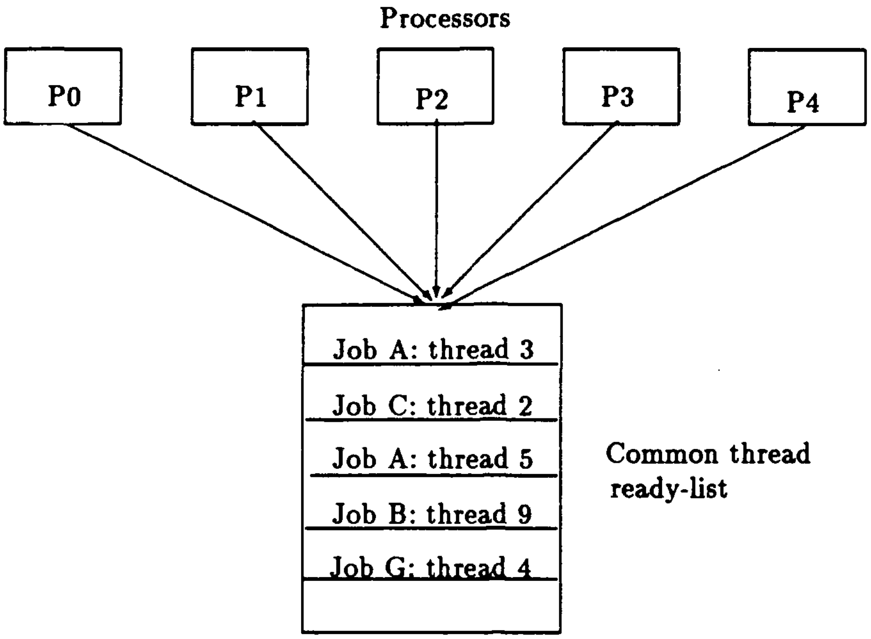


Figure 2: One-level thread scheduling

can be more dramatic. In the previous section, we saw how a coarse-grained program can be slowed down by a factor of two when the number of processors is decreased from five to four. That program exemplified a problem that occurs more generally with preemption and barrier-based synchronization. The program had an implicit barrier, which was the final instruction in the phase. Until all threads reached that instruction, the program could not continue. When one processor was removed, it took twice as long to reach the barrier because not all threads within the job could make progress at an equal rate.

Preemptive multiprocessor scheduling also affects program performance when locks are used, but for a different reason than with barriers. Suppose a thread holding a lock while in a critical section is unexpectedly preempted by the operating system. The lock will remain held until the thread is rescheduled. As threads on other processors try to acquire the lock, they will find it held and be forced to block. It is even possible that, as more threads block waiting for the lock to be freed, the number of that job's runnable threads drops to zero and the application can make no progress until the preempted thread is rescheduled. The overhead of this unnecessary blocking and unblocking slows down the program's execution.

In the previous section, we saw how fine-grained parallelism can improve a program's performance by increasing the chance that a processor will find another runnable thread when its current thread blocks. Unfortunately, a fine-grained parallel program that "packs" the ready-list interacts

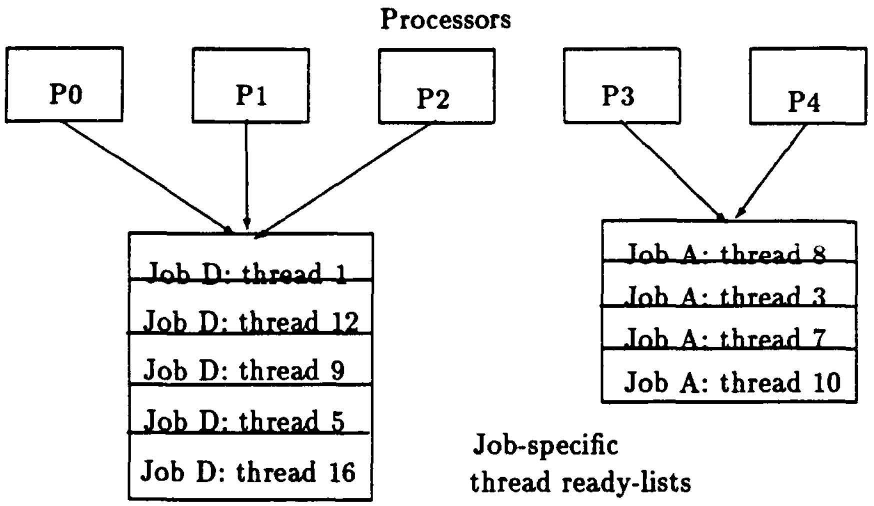


Figure 3: Two-level thread scheduling

badly with the behavior of a one-level scheduler. In particular, when a program's thread blocks in the kernel on an I/O request, the parallelism of the program can only be maintained if the kernel can schedule another of the program's threads in place of the one that blocked. This benefit, though, comes at the cost of increased preemption activity and diminished overall performance.

The problems of one-level scheduling are addressed by two-level schedulers. With a two-level scheduler, processors are first assigned to a job, and then threads within that job are executed only on the assigned processors. Each job has its own ready-list, which is used only by the job's processors, as shown in Figure 3. Thread preemption may no longer be necessary with a two-level scheduler since a preempted thread will only be replaced by another thread from the same job. Further, for long intervals, a processor runs only threads from the same application, so the cost of switching between threads is kept low.

In a two-level scheduling system, processors can be allocated to jobs either statically or dynamically. A static two-level scheduler never changes the number of processors given to a job from its initial allocation; if some of those processors are needed by another job, the operating system must preempt all of the job's processors. A dynamic scheduler can adapt the number of processors assigned to each job according to changing conditions.

Dynamic two-level scheduling can give better performance, because it overlaps periods of poor parallelism in one job with periods of high parallelism in another. One difficulty with a dynamic scheduler is that it requires more information from an application describing the current processor requirements. As a result, though, dynamic scheduling can also more easily handle changes in the number of running jobs. For example, when a job finishes, its processors can be re-allocated to a running job whose parallelism is increasing. To avoid the problems of one-level scheduling, though, it is crucial that the operating system coordinate with each application when it needs to preempt processors (e.g., to avoid preempting a processor when it would seriously affect performance). A dynamic scheduler always has the option, when it needs processors and no application has any available, of reverting to a static policy.

3.2.3 Kernel- vs. User-Level Thread Management

Processor scheduling controls the allocation of processors to jobs. The operating system must be responsible for processor scheduling because processors are a hardware resource and shifting a processor from one job to another involves updating per-processor address space hardware registers. Spawning a thread so that it runs on an already allocated processor, however, does not require modifying privileged state. Thus, thread management and scheduling within a job can be done entirely by the application instead of by the operating system. In this case, thread management operations can be implemented in an application-level library. The library creates virtual processors using the operating system's processor scheduling interface, and schedules the application's threads on top of these virtual processors.

Unlike processor allocation, where a single system-wide scheduling policy can be used, thread scheduling policies benefit from being application-specific. Some applications perform well if their threads are scheduled according to some fixed policy, such as first-in-first-out or last-in-first-out, but others need to schedule threads according to fixed, or even dynamically changing priorities. For example, consider a parallel simulation where each simulation object is represented by its own thread. Different objects become sequential bottlenecks at different times in the simulation; the amount of parallelism can be increased by preferentially scheduling these objects' threads.

It is difficult to provide sufficient thread scheduling flexibility with kernel-level threads. While the kernel could define an interface that allows each application to select its thread scheduling policy, it is unlikely that the system designer could foresee all possible application needs.

Thread management involves more than scheduling. A tradeoff exists between user- and kernel-level thread management. A user-level implementation provides more flexibility and better performance; implementing threads in the kernel guarantees a uniformity that eases the integration of threads with system tools.

The downside of having many custom-built thread management systems is that there is no "standard" thread. By implication, a kernel-level thread management system defines a single, system-wide thread model that is used by all applications. Operating systems that support only one thread model, like those that support only one programming language, can more easily provide sophisticated utilities, such as debuggers and performance monitors. These utilities must rely on the abstraction and often the implementation of the thread model, and a single model makes it easier to provide complete versions of these tools since their cost can be amortized over a large number of applications. Peripheral support for multiple models is possible, but expensive.

A standard thread model also makes it possible for applications to use libraries, or "canned" software utilities. In the same sense that a standard procedure calling sequence sacrifices speed for the ability to call into separately compiled modules, a standard thread model allows one utility to call into another since they both share the same synchronization and concurrency semantics.

It is important to point out that two-level scheduling does not imply that threads are implemented at the application level; the job-specific ready queues shown in Figure 3 could be maintained either within the operating system or within the application. Also, a user-level thread implementation does not imply two-level scheduling, even though threads *are* being scheduled by the application. This implication only holds in the absence of multiprogramming, or in cases where processors are explicitly allocated to jobs. For example, a user-level thread implementation built on top of UNIX processes that share memory suffers from the same problems relating to preemption and I/O as do one-level kernel threads because both are scheduled in a job-independent fashion.

3.3 Performance

The performance of thread operations determines the granularity of parallelism that an application can effectively use. If thread operations are expensive, then applications that have inherently fine-grained parallelism must be re-structured (if that is even possible) to reduce the frequency of those operations. As the cost of thread operations begins to approach that of a few procedure calls, several issues become performance-critical that, for slower operations, would merely be second-order effects.

Simplicity in the thread system's implementation is crucial to performance [Anderson et al. 89]. There is a performance advantage to building multiple thread systems, each tuned for a single type of application. Even simple features that are needed by only some applications, such as saving and restoring all floating point registers on a context switch, will markedly affect the performance of applications that do not need the functionality. Each context switch takes only tens of instructions; a feature that adds even a few more instructions must have a large compensating advantage to be worthwhile. For example, the ability to preemptively schedule threads within each job makes the thread management system more sluggish at several levels, because preemption must be disabled (and then reenabled) whenever scheduling decisions are being made. These scheduling decisions are on the critical path of all thread management operations.

Although kernel-level thread management simplifies the generation and maintenance of system tools, it increases the baseline cost of all thread management operations. Just trapping to the operating system can cost as much as the thread operation itself, making a kernel implementation unattractive for high-performance applications. Further, the generality that must be provided by a kernel-level thread scheduler hurts the performance of those applications needing only basic service. Kernel-level threads are less able to "cut corners" by exploiting application-specific knowledge. With a user-level thread system, the thread management system can be stripped down to provide exactly the functions needed by an application and no more. User-level thread operations also avoid the cost of trapping to the kernel.

Other performance issues have less to do with what a thread system does, than with how it goes about doing it. For example, using a centralized ready-list can limit performance for applications that have extremely fine-grained parallelism. The ready-list is a shared data structure that must be locked to prevent it from being modified by multiple processors simultaneously. Even if the ready-list critical sections consist only of simple enqueue and dequeue operations, they can become a sequential bottleneck, since there is little other work involved in spawning/finishing or blocking/unblocking a thread. An application for which thread overhead is twenty percent of the total execution time, and half of that overhead is spent accessing the ready-list, then its maximum speedup (the time of the parallel program on P processors divided by the time of the program on one processor) is limited to ten.

The bottleneck at the ready-list can be relieved by giving each processor its own ready-list. In this way, enqueueing and dequeueing of work can occur in parallel, with each processor using a different data structure. When a processor becomes idle, it checks its own list for work, and if that list is empty, it scans other processors' lists so that the workload remains balanced.

Per-processor ready-lists have another nice attribute: threads can be preferentially scheduled on the processor on which they last ran, thereby preserving cache state. Computer systems use caches to take advantage of the principle of *locality*, which says that a thread's memory references are directed to or near locations that have been recently referenced. By keeping references close to the processor in fast cache memory, the average time to access a memory location can be kept low. On a multiprocessor, a thread that has been re-scheduled on a different processor will initially find fewer of its references in that processor's cache. For some applications, the cost of fetching these references can exceed the processing time of the thread operation that caused the thread to migrate.

The role of spin-waiting as an optimization technique changes in the presence of high-performance

Basic	Mach	Presto	Multilisp
Spawn	thread_create;thread_resume	Thread::new; Thread::start	(future...)
Block	thread_suspend	Thread::sleep	*Touch unresolved future.*
Unblock	thread_resume	Thread::wakeup	*When future is resolved.*
Finish	thread_terminate	Thread::terminate	*Resolve this future.*

Table 1: The Basic Operations of Thread Management Systems

thread operations. If a thread needs to wait for an event, it can block, relinquishing its processor, or spin-wait. A thread must spin-wait for low-level scheduler locks, but in application code a thread should block instead of spin if the event is likely to take longer than the cost of the context switch. Even though context switches can be implemented efficiently, reducing the need to spin-wait, a hidden cost is that context switches also reduce cache locality.

4 Three Contemporary Thread Systems

We now outline three contemporary thread management systems for multiprocessors: Mach, Presto, and Multilisp. The choices made in each system illustrate many of the thread management issues raised in the previous section.

The thread management primitives for each of these systems are shown in Table 1. The table is organized to indicate how the primitives in one system relate to those in the others, as well as those provided by the basic thread interface outlined in Section 2.2.

Mach is an operating system derived from and compatible with 4.3BSD UNIX, but including extensions to support distributed and parallel programming. Mach supports multiple threads within an address space. Its thread management functions are implemented in the Mach kernel. Since Mach's underlying thread implementation is shared by all parallel programs, system services such as debuggers and performance monitors can be economically provided.

Mach's scheduler uses a priority-based one-level scheduling discipline. Because Mach allocates processors to threads in a job-independent fashion, a parallel program running on top of the Mach thread primitives (or even a user-level thread management system based on those primitives) can suffer from anomalous performance profiles due to ill-timed preemptive decisions made by the one-level scheduling system.

Presto is a user-level thread management system implemented on top of Sequent's DYNIX operating system. DYNIX provides a Presto program with a fixed number of UNIX processes that share memory. The Presto run-time system treats these processes as virtual processors and schedules the user's threads among them. Presto's thread interface is nearly identical to Mach's.

Presto is distinguished from most other thread systems in that it is structured for flexibility. Presto is easy to adapt to application-specific needs because it presents a uniform object-oriented interface to threads, synchronization, and scheduling. The object-oriented design of Presto encourages multiple implementations of the thread management functions and so offers the flexibility to efficiently accommodate differing parallel programming needs.

Presto has been tuned to perform well on a multiprocessor; it tries to avoid bottlenecks in the thread management functions through the use of per-processor data structures. Presto does not provide true two-level scheduling, even though the thread management functions (e.g., thread scheduling) are implemented in an application library accessible to the user, DYNIX, the base operating system, schedules the underlying virtual processors (UNIX processes) any way that it chooses. Although a Presto program can request that its virtual processors not be preempted, the operating system offers no solid guarantee. As a result, kernel preemption threatens the performance of Presto programs in the same was as it does Mach programs.

Although Mach and Presto are implemented differently, the interfaces to each represents a similar style of parallel programming in which the programmer is responsible for explicitly spawning new threads of execution *and* for synchronizing their access to shared data. This style is not accidental, but reflects the basic function of the underlying hardware — processors communicating through shared memory. One criticism often made of this style is that it forces the programmer to think about coordinating many concurrent activities, which can be a conceptually difficult task.

Multilisp demonstrates how thread support can be integrated into a programming language in order to simplify writing parallel programs. In Multilisp, a multiprocessor extension to LISP, the basic concurrency mechanism is the `future`, which is a reference to a data value that has not yet been computed. The `future` operator can be included in any Multilisp expression to spawn a new thread which computes the value of the expression in parallel. Once the value has been computed, the future *resolves* to that value. In the meantime, any thread that tries to use the future's value in an expression automatically blocks until the future is resolved. The language support provided by Multilisp can be implemented on top of a system like Mach or Presto using locks and condition variables.

With Multilisp, the programmer does not need to include any synchronization code beyond the future operator; the Multilisp interpreter keeps track of which futures remain unresolved. By contrast, using the Mach or Presto thread primitives, the programmer must add calls to the appropriate synchronization primitives wherever the data is needed. Multilisp, like Presto, uses per-processor ready-lists to reduce contention in scheduling operations.

5 Summary

This paper has examined some of the key issues in thread management for shared-memory multiprocessors.

Shared-memory multiprocessors are now commonplace in both commercial and research computing. These systems can easily be used to increase throughput for multiprogrammed sequential jobs. However, their greatest potential – as yet not fully realized – is for accelerating the execution of single, parallelized programs.

We have seen that traditional operating system processes are insufficient for expressing other than coarse-grained parallelism. A thread mechanism, constructed at the operating system level, the user level, or both, can help to remedy this problem by reducing the cost of creating and controlling parallelism.

As programmers make use of ~~finer-grain~~ed parallelism, the design and implementation of the thread management system becomes increasingly crucial. Modern thread management systems must address the programmer interface, the operating system interface, and performance optimizations; language support and scheduling techniques for multiprogrammed multiprocessors are two areas that require further research.

References

[Anderson et al. 89] Anderson, T. E., Lazowska, E. D., and Levy, H. M. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. In *1989 ACM SIGMETRICS and Performance '89 Conference on Measurement and Modeling of Computer Systems*, pages 49–60, May 1989.

[Bershad et al. 88] Bershad, B., Lazowska, E., and Levy, H. PRESTO: A System for Object-Oriented Parallel Programming. *Software Practice and Experience*, 18(8):713–732, August 1988.

[Dijkstra 68] Dijkstra, E. W. Cooperating Sequential Processes. In *Programming Languages*, pages 43–112. Academic Press, 1968.

[Encore 86] Encore Computer Corporation. *UMAX 4.2 Programmer's Reference Manual*, 1986.

[Halstead 85] Halstead, R. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transaction on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Hoare 74] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoare 78] Hoare, C. A. R. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Redell et al. 80] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[Sequent 88] Sequent Computer Systems, Inc. *Symmetry Technical Summary*, 1988.

[Tevanian et al. 87] Tevanian, A., Rashid, R. F., Golub, D. B., Black, D. L., Cooper, E., and Young, M. W. Mach Threads and the Unix Kernel: The Battle for Control. In *Proceedings of the 1987 USENIX Summer Conference*, pages 185–197, 1987.

[Thacker et al. 88] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

Lightweight Remote Procedure Call

Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Lightweight Remote Procedure Call (LRPC) is a communication facility designed and optimized for communication between protection domains on the same machine.

In contemporary small-kernel operating systems, existing RPC systems incur an unnecessarily high cost when used for the type of communication that predominates — between protection domains on the same machine. This cost leads system designers to coalesce weakly-related subsystems into the same protection domain, trading safety for performance. By reducing the overhead of same-machine communication, LRPC encourages both safety and performance.

LRPC combines the control transfer and communication model of capability systems with the programming semantics and large-grained protection model of RPC. LRPC achieves a factor of three performance improvement over more traditional approaches based on independent threads exchanging messages, reducing the cost of same-machine communication to nearly the lower bound imposed by conventional hardware.

LRPC has been integrated into the Taos operating system of the DEC SRC Firefly multiprocessor workstation.

1 Introduction

This paper describes Lightweight Remote Procedure Call (LRPC), a communication facility designed and optimized for communication between protection domains on the same machine.

LRPC combines the control transfer and communication model of capability systems with the programming semantics and large-grained protection model of RPC. For the common case of same-machine communication passing small, simple arguments, LRPC achieves a factor of three performance improvement over more traditional approaches.

The granularity of the protection mechanisms used by an operating system has a significant impact on the system's design and use. Some operating systems [Mealy et al. 66, Ritchie & Thompson 74] have large, monolithic kernels insulated from user programs by simple hardware boundaries. Within the operating system itself, though, there are no protection boundaries. The lack of strong firewalls, combined with the size and complexity typical of a monolithic system, make these systems difficult to modify, debug and validate. Further, the shallowness of the protection hierarchy (typically only two levels) makes the underlying hardware directly vulnerable to a large mass of complicated operating system software.

Capability systems supporting *fine-grained* protection were suggested as a solution to the problems of large-kernel operating systems [Dennis & Van Horn 66]. In a capability system, each fine-grained object exists in its own protection domain, but all live within a single name or address space. A process in one domain can act on an object in another only by making a *protected procedure call*, transferring control to the second domain. Parameter passing is simplified by the existence of a global name space containing all objects. Unfortunately, many found it difficult to efficiently implement and program systems that had such fine-grained protection.

In contrast to the fine-grained protection of capability systems, some distributed computing environments rely on relatively *large-grained* protection mechanisms: protection boundaries are defined by machine boundaries [Redell et al. 80]. Remote Procedure Call (RPC) [Birrell & Nelson 84] facilitates the placement of subsystems onto separate machines. Subsystems present themselves to one another in terms of interfaces implemented by servers. The absence of a global address space is ameliorated by automatic stub generators and sophisticated run-time libraries that can transfer arbitrarily complex arguments in messages. RPC is a system structuring and programming style that has become widely successful, enabling efficient and convenient communication across machine boundaries.

Small-kernel operating systems have borrowed the

This material is based on work supported by the National Science Foundation (Grants CCR-8619663, CCR-8700106 and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program). Anderson was supported by an IBM Graduate Fellowship Award, and Bershad was supported by an AT&T Ph.D. Scholarship.

large-grained protection and programming models used in distributed computing environments and have demonstrated these to be appropriate for managing subsystems, even those not primarily intended for remote operation [Rashid 86]. In these small-kernel systems, separate components of the operating system can be placed in disjoint domains (or address spaces), with messages used for all inter-domain communication. The advantages of this approach include modular structure, easing system design, implementation, and maintenance; failure isolation, enhancing debuggability and validation; and transparent access to network services, aiding and encouraging distribution.

In addition to the large-grained protection model of distributed computing systems, small-kernel operating systems have adopted their control transfer and communication models — independent threads exchanging messages containing (potentially) large, structured values. In this paper, though, we show that most communication traffic in operating systems is (1) between domains on the same machine (*cross-domain*), rather than between domains located on separate machines (*cross-machine*), and (2) simple rather than complex. Cross-domain communication dominates because operating systems — even those supporting distribution — localize processing and resources to achieve acceptable performance at reasonable cost for the most common requests. Most communication is simple because complex data structures are concealed behind abstract system interfaces — communication tends to involve only handles to these structures and small value parameters (*booleans, integers, etc.*).

Although the conventional message-based approach can serve the communication needs of both local and remote subsystems, it violates a basic tenet of system design by failing to isolate the common case [Lampson 84]. A cross-domain procedure call *can* be considerably less complex than its cross-machine counterpart, yet conventional RPC systems have not fully exploited this fact. Instead, local communication is treated as an instance of remote communication, and simple operations are considered in the same class as complex ones.

Because the conventional approach has high overhead, today's small-kernel operating systems have suffered from a loss in performance or a deficiency in structure or both. Usually structure suffers most; logically separate entities are packaged together into a single domain, increasing its size and complexity. Such aggregation undermines the primary reasons for building a small-kernel operating system. The Lightweight Remote Procedure Call facility that we describe in this paper arises from these observations.

LRPC achieves a level of performance for cross-domain communication that is significantly better than conventional RPC systems while still retaining their qualities of safety and transparency. Four techniques contribute to the performance of LRPC:

- *Simple control transfer:* The client's thread executes the requested procedure in the server's domain.

- *Simple data transfer:* The parameter passing mechanism is similar to that used by procedure call. A shared argument stack, accessible to both client and server, can often eliminate redundant data copying.

- *Simple stubs:* LRPC uses a simple model of control and data transfer, facilitating the generation of highly optimized stubs.

- *Design for concurrency:* LRPC avoids shared data structure bottlenecks and benefits from the speedup potential of a multiprocessor.

We have demonstrated the viability of LRPC by implementing and integrating it into Taos, the operating system for the DEC SRC Firefly multiprocessor workstation [Thacker et al. 88]. The simplest cross-domain call using LRPC takes 157 microseconds on a single C-VAX processor. By contrast, SRC RPC, the Firefly's native communication system [Schroeder & Burrows 89], takes 464 microseconds to do the same call; though SRC RPC has been carefully streamlined and outperforms peer systems, it is a factor of three slower than LRPC. The Firefly virtual memory and trap handling machinery limit the performance of a safe cross-domain procedure call to roughly 109 microseconds; LRPC adds only 48 microseconds of overhead to this lower bound.

The remainder of this paper discusses LRPC in more detail. Section 2 describes the use and performance of RPC in existing systems, offering motivation for a more lightweight approach. Section 3 describes the design and implementation of LRPC. Section 4 discusses its performance, and section 5 addresses some of the concerns that arise when integrating LRPC into a serious operating system.

2 The Use and Performance of RPC Systems

In this section, using measurements from three contemporary operating systems, we show that only a small fraction of RPCs are *truly* remote, and that large or complex parameters are rarely passed during non-remote operations. We also show the disappointing performance of cross-domain RPC in several systems. These results demonstrate that simple, cross-domain calls represent the common case and can be well-served by optimization.

2.1 Frequency of Cross-Machine Activity

We examined three operating systems to determine the relative frequency of cross-machine activity.

- The V System

  In V [Cheriton 88], a highly decomposed system, only the basic message primitives (Send, Receive, etc.) are accessed directly through kernel traps. All other system functions are accessed by sending messages to the appropriate server. Concern for efficiency, though, has forced the implementation of many of these servers down into the kernel.

  In an instrumented version of the V system, Williamson found that 97% of calls crossed protection, but not machine, boundaries [Williamson 89]. Williamson's measurements include message traffic to kernel-resident servers.

- Taos

  Taos, the Firefly operating system, is divided into two major pieces A medium-sized privileged kernel accessed through traps is responsible for thread scheduling, virtual memory. and device access. A second, multi-megabyte domain accessed through RPC implements the remaining pieces of the operating system (domain management, local and remote file systems, window management, network protocols, etc.). Taos does not cache remote files, but each Firefly node is equipped with a small disk for storing local files to reduce the frequency of network operations.

  We measured activity on a Firefly multiprocessor workstation connected to a network of other Fireflies and a remote file server. During one five-hour work period, we counted 344,888 local RPC calls, but only 18,366 network RPCs. Cross-machine RPCs thus accounted for only 5.3% of all communication activity.

- UNIX+NFS

  In UNIX, a large-kernel operating system, all local system functions are accessed through kernel traps. RPC is used only to access remote file servers. Although a UNIX system call is not implemented as a cross-domain RPC, in a more decomposed operating system most calls would result in at least one such RPC.

  On a diskless Sun 3 workstation running Sun UNIX+NFS [Sandberg et al. 85], during a period of four days we observed over 100 million operating system calls, but fewer than one million RPCs to file servers. Inexpensive system calls, encouraging frequent kernel interaction, and file caching, eliminating many calls to remote file servers, are together responsible for the relatively small number of cross-machine operations.

Operating System	**Percentage of Operations That Cross Machine Boundaries**
V	3%
Taos	5.3%
Sun Unix+NFS	0.6%

Table 1: Frequency of Remote Activity

Table 1 summarizes our measurements of these three systems. Our conclusion is that most calls go to targets on the same node. While measurements of systems taken under different workloads will demonstrate different percentages, we believe that cross-domain activity, rather than cross-machine activity, will dominate. Because a cross-machine RPC is slower than even a slow *cross-domain* RPC, system builders have an incentive to avoid network communication. This incentive manifests itself in the many different caching schemes used in distributed computing systems.

2.2 Parameter Size and Complexity

The second part of our RPC evaluation is an examination of the size and complexity of cross-domain procedure calls. Our analysis considers both the dynamic and static usage of SRC RPC as used by the Taos operating system and its clients. The size and maturity of the system make it a good candidate for study — our version includes 28 RPC services defining 366 procedures involving over 1000 parameters.

We counted 1,487,105 cross-domain procedure calls during one four-day period. Although 112 different procedures were called, 95% of the calls were to ten procedures, and 75% were to just three. None of the stubs for these three were required to marshal complex arguments — byte copying was sufficient to transfer the data between domains. [1]

In the same four days, we also measured the number of bytes transferred between domains during cross-domain calls. Figure 1, a histogram and cumulative distribution of this measure, shows that the most frequently occurring calls transfer fewer than 50 bytes, and a majority transfer fewer than 200.

Statically, we found that four out of five parameters were of fixed size known at compile time; sixty-five percent were four bytes or fewer. Two-thirds of all procedures passed only parameters of fixed size, and sixty percent transferred 32 or fewer bytes. No data types were recursively defined so as to require recursive marshaling (such as linked lists or binary trees). Recursive types *were* passed through RPC interfaces, but these were marshaled by system library procedures, rather than by machine-generated code.

These observations indicate that simple byte copying is usually sufficient for transferring data across system interfaces, and that the majority of interface procedures move only small amounts of data.

---

[1] SRC RPC maps domain-specific pointers into and out of network-wide unique representations, enabling pointers to be passed back and forth across an RPC interface. The mapping is done by a simple table-lookup, and was necessary for two of the top three procedures.

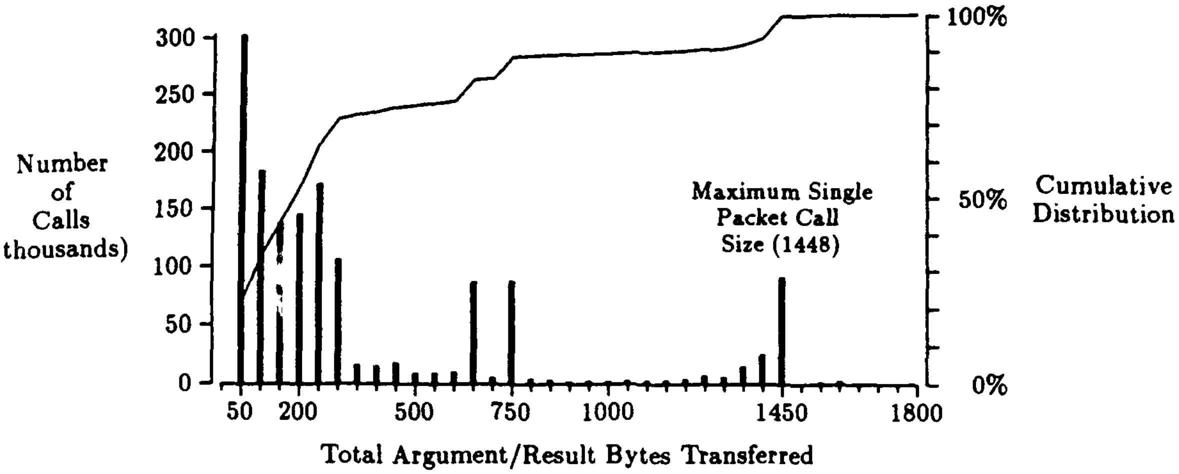


Figure 1: RPC Size Distribution

Others have noticed that most interprocess communication is simple, passing mainly small parameters [Cook 78, Cheriton 88, Karger 89], and some have suggested optimizations for this case. V, for example, uses a message protocol that has been optimized for fixed-sized messages of 32 bytes. Karger describes compiler-driven techniques for passing parameters in registers during cross-domain calls. These optimizations, although sometimes effective, only partially address the performance problems of cross-domain communication.

2.3 The Performance of Cross-Domain RPC

In existing RPC systems, cross-domain calls are implemented in terms of the facilities required by cross-machine ones. Even through extensive optimization, good cross-domain performance has been difficult to achieve. Consider the Null procedure call that takes no arguments, returns no values and does nothing:

```
PROCEDURE Null(); BEGIN RETURN END Null;
```

The theoretical minimum time to invoke Null() as a cross-domain operation involves one procedure call, followed by a kernel trap and change of the processor's virtual memory context on call, and then a trap and context change again on return. The difference between this theoretical minimum call time and the actual Null call time reflects the overhead of a particular RPC system. Table 2 shows this overhead for six systems. The data in Table 2 comes from measurements of our own and from published sources [Fitzgerald 86, Tzou & Anderson 88, van Renesse et al. 88].

The high overheads revealed by Table 2 can be attributed to several aspects of conventional RPC:

- *Stub overhead:* Stubs provide a simple procedure call abstraction, concealing from programs the interface to the underlying RPC system. The distinction between cross-domain and cross-machine calls is usually made transparent to the stubs by lower levels of the RPC system. This results in an interface and execution path that are general but infrequently needed. For example, it takes about 70 microseconds to execute the stubs for the Null procedure call in SRC RPC. Other systems have comparable times.
- *Message buffer overhead:* Messages need to be allocated and passed between the client and server domains. Cross-domain message transfer can involve an intermediate copy through the kernel, requiring four copy operations for any RPC (two on call, two on return).
- *Access Validation:* The kernel needs to validate the message sender on call and then again on return.
- *Message transfer:* The sender must enqueue the message, which must later be dequeued by the receiver. Flow-control of these queues is often necessary.
- *Scheduling:* Conventional RPC implementations bridge the gap between *abstract* and *concrete* threads. The programmer's view is one of a single, abstract thread crossing protection domains, while the underlying control transfer mechanism involves concrete threads fixed in their own domain signalling one another at a rendezvous. This indirection can be slow, as the scheduler must manipulate system data structures to block the client's concrete thread and then select one of the server's for execution.
- *Context switch:* There must be a virtual memory context switch from the client's domain to the server's on call, and then back again on return.

System	Processor	Null (Theoretical Minimum)	Null (Actual)	Overhead
Accent	PERQ	444	2300	1856
Taos	Firefly C-VAX	109	464	355
Mach	C-VAX	90	754	664
V	68020	170	730	560
Amoeba	68020	170	800	630
DASH	68020	170	1590	1420

Table 2: Cross-Domain Performance (times are in microseconds)

- *Dispatch:* A receiver thread in the server domain must interpret the message and dispatch a thread to execute the call. If the receiver is self-dispatching, it must ensure that another thread remains to collect messages that may arrive before the receiver finishes to prevent caller serialization.

RPC systems have optimized some of these steps in an effort to improve cross-domain performance. The DASH system [Tzou & Anderson 88] eliminates an intermediate kernel copy by allocating messages out of a region specially mapped into both kernel and user domains. Mach [Jones & Rashid 86] and Taos rely on *handoff scheduling* to bypass the general, slower scheduling path; instead, if the two concrete threads cooperating in a domain transfer are identifiable at the time of the transfer, a direct context switch can be made. In line with handoff scheduling, some systems pass a few, small arguments in registers, thereby eliminating buffer copying and management. [2]

SRC RPC represents perhaps the most ambitious attempt to optimize traditional RPC for swift cross-domain operation. Unlike techniques used in other systems which provide safe communication between mutually suspicious parties, SRC RPC trades safety for increased performance. To reduce copying, message buffers are globally shared across all domains. A single lock is mapped into all domains so that message buffers can be acquired and released without kernel involvement. Further, access validation is not performed on call and return, simplifying the critical transfer path.

SRC RPC runs much faster than other RPC systems implemented on comparable hardware. Nevertheless, SRC RPC still incurs a large overhead due to its use of heavyweight stubs and run-time support, dynamic buffer management, multi-level dispatch, and interaction with global scheduling state.

3 The Design and Implementation of LRPC

The lack of good performance for cross-domain calls has encouraged system designers to coalesce cooperating subsystems into the same domain. Applications use RPC to communicate with the operating system, ensuring protection and failure isolation for users and the collective system. The subsystems themselves, though, grouped into a single protection domain for performance reasons, are forced to rely exclusively on the thin barriers provided by the programming environment for protection from one another. LRPC solves, rather then circumvents, this performance problem in a way that does not sacrifice safety.

The execution model of LRPC is borrowed from protected procedure call. A call to a server procedure is made by way of a kernel trap. The kernel validates the caller, creates a call linkage, and dispatches the client's concrete thread directly to the server domain. The client provides the server with an argument stack as well as its own concrete thread of execution. When the called procedure completes, control and results return through the kernel back to the point of the client's call.

The programming semantics and large-grained protection model of LRPC are borrowed from RPC. Servers execute in a private protection domain, and each exports one or more interfaces, making a specific set of procedures available to other domains. A client *binds* to a server interface before making the first call. The server, by allowing the binding to occur, authorizes the client to access the procedures defined by the interface.

3.1 Binding

At a conceptual level, LRPC binding and RPC binding are similar. Servers export interfaces and clients bind to those interfaces before using them. At a lower-level, however, LRPC binding is quite different due to the high degree of interaction and cooperation that is required of the client, server and kernel.

A server module exports an interface through a clerk in the LRPC run-time library included in every domain. The clerk registers the interface with a name server and awaits import requests from clients. A client binds to a specific interface by making an import call via the kernel. The importer waits while the kernel notifies the server's waiting clerk.

---

[2] Optimizations based on passing arguments in registers exhibit a performance discontinuity once the parameters overflow the registers. The data in Figure 1 indicates that this can be a frequent problem.

The clerk enables the binding by replying to the kernel with a *procedure descriptor list* (PDL) that is maintained by the exporter of every interface. The PDL contains one *procedure descriptor* (PD) for each procedure in the interface. The PD includes an entry address in the server domain, the number of simultaneous calls initially permitted to the procedure by the client, and the size of the procedure's *argument stack* (A-stack) on which arguments and return values will be placed during a call. For each PD, the kernel pair-wise allocates in the client and server domains a number of A-stacks equal to the number of simultaneous calls allowed. These A-stacks are mapped read-write and shared by both domains.

Procedures in the same interface having A-stacks of similar size can share A-stacks, reducing the storage needs for interfaces with many procedures. The number of simultaneous calls initially permitted to procedures that are sharing A-stacks is limited by the total number of A-stacks being shared. This is only a soft limit, though, and Section 5.2 describes how it can be raised.

The kernel also allocates a *linkage record* for each A-stack that is used to record a caller's return address and is accessible only to the kernel. The kernel lays out A-stacks and linkage records in memory in a way such that the correct linkage record can be quickly located given any address in the corresponding A-stack.

After the binding has completed, the kernel returns to the client a Binding Object. The Binding Object is the client's key for accessing the server's interface and must be presented to the kernel at each call. The kernel can detect a forged Binding Object, so clients cannot bypass the binding phase. In addition to the Binding Object, the client receives an A-stack list for each procedure in the interface giving the size and location of the A-stacks that should be used for calls into that procedure.

3.2 Calling

Each procedure in an interface is represented by a stub in the client and server domains. A client makes an LRPC by calling into its stub procedure which is responsible for initiating the domain transfer. The stub manages the A-stacks allocated at bind time for that procedure as a LIFO queue. At call time, the stub takes an A-stack off the queue, pushes the procedure's arguments onto the A-stack, puts the address of the A-stack, the Binding Object and a procedure identifier into registers, and traps to the kernel. In the context of the client's thread, the kernel

- verifies the Binding and procedure identifier
- verifies the A-stack and locates the corresponding linkage
- ensures that no other thread is currently using that A-stack/linkage pair
- records the caller's return address and current stack pointer in the linkage
- pushes the linkage onto the top of a stack of linkages kept in the thread's control block[3]
- finds an execution stack (*E-stack*) in the server's domain
- updates the thread's user stack pointer to run off of the new E-stack
- reloads the processor's virtual memory registers with those of the server domain
- performs an upcall [Clark 85] into the server's stub at the address specified in the PD for the requested procedure.

Arguments are pushed onto the A-stack according to the calling conventions of Modula2+ [Rovner et al. 85]. Since the A-stack is mapped into the server's domain, the server procedure can directly access the parameters as though it had been called directly. It's important to note that this optimization relies on a calling convention that uses a separate argument pointer. In a language environment that required arguments to be passed on the E-stack, this optimization would not be possible.

The server procedure returns through its own stub, which initiates the return domain transfer by trapping to the kernel. Unlike the call, which required presentation and verification of the Binding Object, procedure identifier and A-stack, this information, contained at the top of the linkage stack referenced by the thread's control block, is implicit in the return. There is no need to verify the returning thread's right to transfer back to the calling domain since it was granted at call time. Further, since the A-stack contains the procedure's return values, and the client specified the A-stack on call, no explicit message needs to be passed back.

If any parameters are passed by reference, the client stub copies the referent onto the A-stack. The server stub creates a reference to the data and places the reference on its private E-stack before invoking the server procedure. The reference must be recreated to prevent the caller from passing in a bad address. The data, though, is not copied and remains on the A-stack.

Privately mapped E-stacks enable a thread to safely cross between domains. Conventional RPC systems provide this safety by implication, deriving separate stacks from separate threads. LRPC excises this level of indirection, dealing directly with less weighty stacks.

A low-latency domain transfer path requires that E-stack management incur little call-time overhead. One way to achieve this is to statically allocate E-stacks at bind time and to permanently associate each with an A-stack. Unfortunately, E-stacks can be large (tens of kilobytes) and must be managed conservatively; otherwise a server's address space could be exhausted by just a few clients.

---

[3] The stack is necessary so that a thread can be involved in more than one cross-domain procedure call at a time.

Rather than statically allocating E-stacks, LRPC delays the A-stack/E-stack association until it is needed; that is, until a call is made with an A-stack not having an associated E-stack. When this happens, the kernel checks if there is an E-stack already allocated in the server domain, but currently unassociated with any A-stack. If so, the kernel associates the E-stack with the A-stack. Otherwise, the kernel allocates an E-stack out of the server domain and associates it with the A-stack. When the call returns, the E-stack and A-stack remain associated with one another so that they might be used together soon for another call (A-stacks are LIFO managed by the client). Whenever the supply of E-stacks for a given server domain runs low, the kernel reclaims those associated with A-stacks that have not been recently used.

3.3 Stub Generation

Stubs bridge the gap between procedure call, the communication model used by the programmer, and domain transfer, the execution model of LRPC. A procedure is represented by a call stub in the client's domain and an entry stub in the server's. Every procedure declared in an LRPC interface defines the terminus of a three-layered communication protocol: end-to-end, described by the calling conventions of the programming language and architecture; stub-to-stub, implemented by the stubs themselves; and domain-to-domain, implemented by the kernel.

LRPC stubs blur the boundaries between the protocol layers to reduce the cost of crossing between them. Server entry stubs are invoked directly by the kernel on a transfer; no intermediate message examination and dispatch is required. The kernel primes E-stacks with the initial call frame expected by the server's procedure, enabling the server stub to branch to the first instruction of the procedure. As a result, a simple LRPC needs only one formal procedure call (into the client stub), and two returns (one out of the server procedure and one out of the client stub).

The LRPC stub generator produces run-time stubs in assembly language directly from Modula2+ definition files. The use of assembly language is possible because of the simplicity and stylized nature of LRPC stubs, which consist mainly of move and trap instructions. The LRPC stubs have shown a factor of four performance improvement over Modula2+ stubs created by the SRC RPC stub generator.

Since the stubs are automatically generated, the only maintenance concerns arising from this use of assembly language are related to the portability of the stub generator (the stubs themselves are not portable, but we don't consider this to be an issue). Porting the stub generator to work on a different machine architecture should be a straightforward task, although we have not yet had any reason to do so.

The stub generator emits Modula2+ code for more complicated, but less frequently traveled execution paths, such as those dealing with binding, exception handling, and call failure. Calls having complex or heavyweight parameters — linked lists or data that must be made known to the garbage collector — are handled with Modula2+ marshaling code. LRPC stubs become more like conventional RPC stubs as the overhead of dealing with the complicated data types increases. This shift occurs at compile-time, eliminating the need to make run-time decisions.

3.4 LRPC on a Multiprocessor

The existence of shared-memory multiprocessors has influenced the design of LRPC. Multiple processors can be used to achieve a higher call throughput and lower call latency than is possible on a single processor.

LRPC increases throughput by minimizing the use of shared data structures on the critical domain transfer path. Each A-stack queue is guarded by its own lock, and queuing operations take less than 2% of the total call time. No other locking occurs, so there is little interference when calls occur simultaneously.

Multiple processors are used to reduce LRPC latency by caching domain contexts on idle processors. As we show in Section 4, the context switch that occurs during an LRPC is responsible for a large part of the transfer time. This time is due partly to the code required to update the hardware's virtual memory registers, and partly to the extra memory fetches that occur as a result of invalidating the translation lookaside buffer (TLB).

LRPC reduces context-switch overhead by caching domains on idle processors. When a call is made, the kernel checks for a processor idling in the context of the server domain. If one is found, the kernel exchanges the processors of the calling and idling threads, placing the calling thread on a processor where the context of the server domain is already loaded; the called server procedure can then execute on that processor without requiring a context switch. The idling thread continues to idle, but on the client's original processor in the context of the client domain. On return from the server, a check is also made. If a processor is idling in the client domain (likely for calls that return quickly), then the processor exchange can be done again.

If no idle domain can be found on call or return, then a single-processor context switch is done. For each domain, the kernel keeps a counter indicating the number of times that a processor idling in the context of that domain was needed but not found. The kernel uses these counters to prod idle processors to spin in domains showing the most LRPC activity.

The high cost of frequent domain crossing can also be reduced by using a TLB that includes a process tag. For multiprocessors without such a tag, domain-caching can often achieve the same result for commonly called servers. Even with a tagged TLB, a single-processor domain switch still requires that hardware mapping registers be modified on the critical transfer path; domain

Operation	LRPC	Message Passing	Restricted Message Passing
call (mutable parameters)	A	ABCE	ADE
call (immutable parameters)	AE	ABCE	ADE
return	F	BCF	BF

Code	Copy Operation
A	copy from client stack to message (or A-stack)
B	copy from sender domain to kernel domain
C	copy from kernel domain to receiver domain
D	copy from sender/kernel space to receiver/kernel domain
E	copy from message (or A-stack) into server stack
F	copy from message (or A-stack) into client's results

Table 3: Copy Operations For LRPC Vs. Message-Based RPC

caching does not. Finally, domain caching preserves per-processor locality across calls — a performance consideration for systems having low tolerance for sudden shifts in locality.

Using idle processors to decrease operating system latency is not a new idea. Both Amoeba and Taos cache recently blocked threads on idle processors to reduce wakeup latency. LRPC generalizes this technique by caching domains, rather than threads. In this way, any thread that needs to run in the context of an idle domain can do so quickly, not just the thread that ran there most recently.

3.5 Argument Copying

Consider the path taken by a procedure's argument during a traditional cross-domain RPC. An argument, beginning with its placement on the stack of the client stub, is copied 4 times — from the stub's stack to the RPC message, from the message in the client's domain to one in the kernel's, from the message in the kernel's domain to one in the server's, and from the message to the server's stack. The same argument in an LRPC can be copied only once: from the stack of the client stub to the shared A-stack from which it can be used by the server procedure.

Pair-wise allocation of A-stacks enables LRPC to copy parameters and return values only as many times as are necessary to ensure correct and safe operation. Protection from third-party domains is guaranteed by the pair-wise allocation that provides a private channel between the client and server. It is still possible for a client or server to asynchronously change the values of arguments in an A-stack once control has transferred across domains. The copying done by message-based RPC prevents such changes, but often at a higher cost than necessary. LRPC, by considering each argument individually, avoids extra copy operations by taking advantage of argument passing conventions, by exploiting a value's correctness semantics, and by combining the copy into a check for the value's integrity.

In most procedure call conventions, the destination address for return values is specified by the caller. During the return from an LRPC, the client stub copies returned values from the A-stack into their final destination. No added safety comes from first copying these values out of the server's domain into the client's, either directly or by way of the kernel.

Parameter copying can also be avoided by recognizing situations in which the actual value of the parameter is unimportant to the server. This occurs when parameters are processed by the server without interpretation. For example, the *Write* procedure exported by a file server takes an array of bytes to be written to disk. The array itself is not interpreted by the server, which is made no more secure by an assurance that the bytes won't change during the call. Copying is unnecessary in this case. These types of arguments can be identified to the LRPC stub generator.

Finally, concern for type safety motivates explicit argument copying in the stubs, rather than wholesale message copying in the kernel. In a strongly-typed language, such as Modula2+, actual parameters must conform to the types of the declared formals; for example, the Modula2+ type CARDINAL is restricted to the set of positive integers — a negative value will result in a run-time error when the value is used. A client could crash a server by passing it an unwanted negative value. To protect itself, the server must check type-sensitive values for conformancy before using them. Folding this check into the copy operation can result in less work than if the value is first copied by the message system and then later checked by the stubs.

Table 3 shows how the use of A-stacks in LRPC can affect the number of copying operations. For calls where parameter immutability is important, and for those where it isn't, we compare the behavior of LRPC against the traditional message-passing approach, and

Test	Description	LRPC/MP	LRPC	Taos
Null	the Null cross-domain call	125	157	464
Add	a procedure taking two 4-byte arguments and returning one 4-byte argument	130	164	480
BigIn	a procedure taking one 200-byte argument	173	192	539
BigInOut	a procedure taking and then returning one 200-byte argument	219	227	636

Table 4: LRPC Performance of Four Tests (in microseconds)

against a more restricted form of message-passing used in the DASH system. In the restricted form, all message buffers on the system are allocated from a specially mapped region that enables the kernel to copy messages directly from the sender's domain into the receiver's, avoiding an intermediate kernel copy.

In Table 3, we assume that the server places the results directly into the reply message. If this isn't the case (i.e., messages are managed as a scarce resource), then one more copy from the server's results into the reply message is needed. Even when the immutability of parameters is important, LRPC performs fewer copies (3) than either message passing (7) or restricted message passing (5).

For passing large values, copying concerns become less important, since by-value semantics can be achieved through virtual memory operations. But, for the more common case of small- to medium-sized values, eliminating copy operations is crucial to good performance when call latency is on the order of only 100 *instructions*.

LRPC's A-stack/E-stack design offers both safety and performance. While our implementation demonstrates the performance of this design, the Firefly operating system does not yet support pair-wise shared memory. Our current implementation places A-stacks in globally shared virtual memory. Since mapping is done at bind time, an implementation using pair-wise shared memory would have identical performance, but greater safety.

4 The Performance of LRPC

To evaluate the performance of LRPC, we used the four tests shown in Table 4. These tests were run on the C-VAX Firefly using LRPC and Taos RPC. The Null call provides a baseline against which we can measure the added overhead of LRPC. The procedures Add, BigIn, and BigInOut represent calls having "typical" parameter sizes.

Table 4 shows the results of these tests when performed on a single node. The measurements were made by performing 100,000 cross-domain calls in a tight loop, computing the elapsed time, and then dividing by 100,000. The table shows two times for LRPC. The first, listed as "LRPC/MP," uses the idle processor optimization described in Section 3.4. The second, shown as "LRPC," executes the domain switch on a single processor; it is roughly 3 times faster than SRC RPC, which also uses only one processor.

Table 5 shows a detailed cost breakdown for the serial (1-processor) Null LRPC on a C-VAX. This table was produced from a combination of timing measurements and hand calculations of TLB misses. The code to execute a Null LRPC consists of 120 instructions that require 157 microseconds to execute. The column labeled "Minimum" in Table 5 is a timing breakdown for the theoretically minimum cross-domain call (one procedure call, two traps and two context switches). The column labeled "LRPC Overhead" shows the additional time required to execute the call and return operations described in Section 3.2 and is the cost of our implementation. For the Null call, approximately 18 microseconds are spent in the client stub and 3 in the server's. The remaining 27 microseconds of overhead are spent in the kernel, and go towards binding validation and linkage management. Most of this takes place during the call, as the return path is simpler.

Operation	Minimum	LRPC Overhead
Modula2+ Procedure Call	7	
Two Kernel Traps	36	
Two Context Switches	66	
Stubs		21
Kernel Transfer		27
TOTAL	109	48

Table 5: Breakdown of Time (in microseconds) for Single Processor Null LRPC

Approximately 25% of the time used by the Null LRPC is due to TLB misses that occur during virtual address translation. A context switch on a C-VAX requires the invalidation of the TLB, and each subsequent TLB miss increases the cost of a memory reference by about .9 microseconds. Anticipating this, the data structures and control sequences of LRPC were designed to minimize TLB misses. Even so, we estimate that 43 TLB misses occur during the Null call.

Section 3.4 stated that LRPC avoids locking shared data during call and return in order to remove contention on shared-memory multiprocessors. This is demonstrated by Figure 2, which shows call throughput as a function of the number of processors simultaneously making calls. Domain caching was disabled for

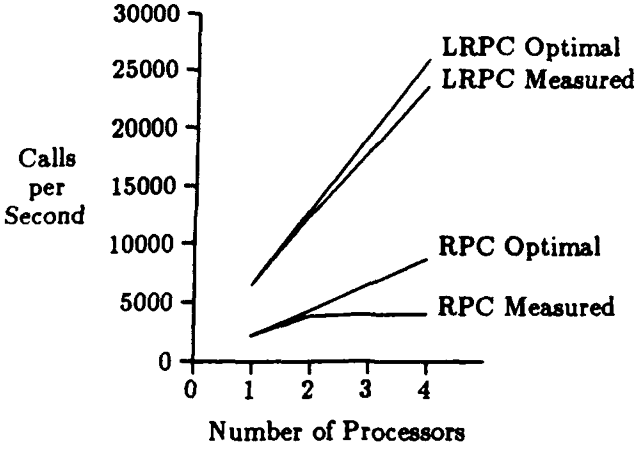


Figure 2: Call Throughput On a Multiprocessor

this experiment — each call required a context switch. A single processor can make about 6300 LRPCs per second, but four processors can make over 23000 calls per second — a speedup of 3.7 and close to the maximum that the Firefly is capable of delivering. These measurements were made on a Firefly having four C-VAX processors and one MicroVaxII I/O processor. Measurements on a five processor MicroVaxII Firefly showed a speedup of 4.3 with 5 processors.

In contrast, the throughput of SRC RPC levels off with two processors at about 4000 calls per second. This limit is due to a global lock that is held during a large part of the RPC transfer path. For a machine like the Firefly, a small scale shared-memory multiprocessor, a limiting factor of two is annoying, but not serious. On shared-memory machines with just a few dozen processors, though, contention on the critical control transfer path would have a greater performance impact.

5 The Uncommon Cases

In addition to working well in the common case, LRPC must work acceptably in the less common ones. This section describes several of these less common cases and explains how they are dealt with by LRPC. This section does not enumerate all possible uncommon cases that must be considered. Instead, by describing just a few, we hope to emphasize that the common-case approach taken by LRPC is flexible enough to accommodate the uncommon cases gracefully.

5.1 Transparency and Cross-Machine Calls

Deciding whether a call is cross-domain or cross-machine is made at the earliest possible moment — the first instruction of the stub. If the call is to a truly remote server (indicated by a bit in the Binding Object), then a branch is taken to a more conventional RPC stub. The extra level of indirection is negligible compared to the overheads that are part of even the most efficient network RPC implementation.

5.2 A-stacks — Size and Number

Procedure Descriptor Lists are defined during the compilation of an interface. The stub generator reads each interface and determines the number and size of the A-stacks for each procedure. The number defaults to five, but can be overridden by the interface writer. When the size of each of a procedure's arguments and return values are known at compile time, the A-stack size can be determined exactly. In the presence of variable sized arguments, though, the stub generator uses a default size equal to the Ethernet packet size (this default also can be overridden). Experience has shown, and Figure 1 confirms, that RPC programmers strive to keep the sizes of call and return parameters under this limit. Most existing RPC protocols are built on simple packet exchange protocols, and multi-packet calls have performance problems. In cases where the arguments are too large to fit into the A-stack, the stubs transfer data in a large out-of-band memory segment. Handling unexpectedly large parameters is complicated and relatively expensive, but infrequent.

A-stacks in a single interface are allocated contiguously at bind time to allow for quick validation during a call (a simple range check guarantees their integrity). If the number of pre-allocated A-stacks proves too few, the client can either wait for one to become available (when an earlier call finishes), or allocate more. Waiting is simple, but may not always be appropriate. When further allocation is necessary, it is unlikely that space contiguous to the original A-stacks will be found, but other space can be used. A-stacks in this space, not in the primary contiguous region, will take slightly more time to validate during a call.

5.3 Domain Termination

A domain can terminate at any time, for reasons such as an unhandled exception or a user action (CTRL-C). When a domain terminates, all resources in its possession (virtual address space, open file descriptors, threads, etc.) are reclaimed by the operating system. If the terminating domain is a server handling an LRPC request, the call, completed or not, must return to the client domain. If the terminating domain is a client with a currently outstanding LRPC request to another domain, the outstanding call, when finished, must not be allowed to return to its originating domain.

When a domain is terminated, each Binding Object associated with that domain (either as client or server) is revoked. This prevents any more out-calls from the domain, and prevents other domains from making any more in-calls. All threads executing within the domain are then stopped, and a kernel collector scans all of the domain's threads looking for any that had been running on behalf of an LRPC call; these threads are restarted in the client with a call-failed exception. Finally, the collector scans all Binding Objects held by the terminating domain and invalidates any active linkage records. When a thread returns from an LRPC call, it follows the stack of linkage records referenced by the thread control block, returning to the domain specified in the first valid linkage record. If any invalid linkage records are found on the way, a call-failed exception is raised in the caller. If the stack contains no valid linkage records, the thread is destroyed.

A terminating domain's outstanding threads are not forced to terminate synchronously with the domain. Doing so would require every server procedure to protect the integrity of its critical data structures from external forces, since a mutating thread could be terminated at any time. More generally, LRPC has no way to force a thread to return from an outstanding call. Taos does have an *alert* mechanism which allows one thread to signal another, but the notified thread may choose to ignore the alert. It is therefore possible for one domain to "capture" another's thread and hold it indefinitely. To address this problem, LRPC enables client domains to create a new thread whose initial state is that of the original captured thread as if it had just returned from the server procedure with a call-aborted exception. The captured thread continues executing in the server domain but is destroyed in the kernel when released.

Traditional RPC does not have these problems because the abstract thread seen by the programmer is provided by two concrete threads, one in each of the client and server domains. Because premature domain and call termination are infrequent, LRPC has adopted a "special case" approach for dealing with them.

6 Summary

This paper has described the motivation, design, implementation, and performance of LRPC, a communication facility that combines elements of capability and RPC systems. Our implementation on the Firefly achieves performance that is close to the minimum round-trip cost of transferring control between domains on conventional hardware.

LRPC adopts a common-case approach to communication, exploiting, whenever possible, simple control transfer, simple data transfer, simple stubs, and multiprocessors. In so doing, LRPC performs well for the majority of cross-domain procedure calls by avoiding needless scheduling, excessive run-time indirection, unnecessary access validation, redundant copying, and lock contention. LRPC, nonetheless, is safe and transparent, and represents a viable communication alternative for small-kernel operating systems.

7 Acknowledgements

We would like to thank Guy Almes, David Anderson, Andrew Birrell, Mike Burrows, Dave Cutler, Roy Levin, Mark Lucovsky, Tim Mann, Brian Marsh, Rick Rashid, Dave Redell, Jan Sanislo, Mike Schroeder, Shin-Yuan Tzou, and Steve Wood for discussing with us the issues raised in this paper. We would also like to thank DEC SRC for building and supplying us with the Firefly. It has been a challenge to improve on the excellent performance of SRC RPC, but one made easier by the Firefly's overall structure. One measure of a system's design is how easily a significant piece of it can be changed. We doubt that we could have implemented LRPC as part of any other system as painlessly as we did on the Firefly.

References

[Birrell & Nelson 84] Birrell, A. D. and Nelson, B. J. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Cheriton 88] Cheriton, D. R. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[Clark 85] Clark, D. D. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171–180, December 1985.

[Cook 78] Cook, D. *The Evaluation of a Protection System*. PhD dissertation, Cambridge University, Computer Laboratory, April 1978.

[Dennis & Van Horn 66] Dennis, J. B. and Van Horn, E. C. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[Fitzgerald 86] Fitzgerald, R. P. *A Performance Evaluation of the Integration of Virtual Memory Management and Inter-Process Communication in Accent*. PhD dissertation, Carnegie-Mellon University, October 1986.

[Jones & Rashid 86] Jones, M. B. and Rashid, R. F. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 67–77, October 1986.

[Karger 89] Karger, P. A. Using Registers to Optimize Cross-Domain Call Performance. In *Proceedings of the Third Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[Lampson 84] Lampson, B. W. Hints for Computer System Design. *IEEE Software*, 1(1):11–28, January 1984.

[Mealy et al. 66] Mealy, G., Witt, B., and Clark, W. The Functional Structure of OS/360. *IBM Systems Journal*, 5(1):3–51, 1966.

[Rashid 86] Rashid, R. F. From Rig to Accent to Mach: The Evolution of a Network Operating System. In *Proceeding of ACM/IEEE Computer Society Fall Joint Computer Conference*, November 1986.

[Redell et al. 80] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, pages 81–92, February 1980.

[Ritchie & Thompson 74] Ritchie, D. and Thompson, K. The Unix Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[Rovner et al. 85] Rovner, P., Levin, R., and Wick, J. On Extending Modula-2 For Building Large, Integrated Systems. Technical Report # 3, Digital Equipment Corporation Systems Research Center, Palo Alto, California, January 1985.

[Sandberg et al. 85] Sandberg, R., Goldberg, D., Steve Kleiman, D. W., and Lyon, B. Design and Implementation of the SUN Network Filesystem. In *Proceedings of the 1985 USENIX Summer Conference*, pages 119–130, 1985.

[Schroeder & Burrows 89] Schroeder, M. D. and Burrows, M. Performance of Firefly RPC. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989. To appear in ACM Transactions on Computer Systems, February 1990.

[Thacker et al. 88] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Tzou & Anderson 88] Tzou, S.-Y. and Anderson, D. P. A Performance Evaluation of the DASH Message-Passing System. Technical Report UCB/CSD 88/452, Computer Science Division, University of California, Berkeley, October 1988.

[van Renesse et al. 88] van Renesse, R., van Staveren, H., and Tanenbaum, A. S. Performance of the World's Fastest Distributed Operating System. *Operating Systems Review*, 22(4):25–34, October 1988.

[Williamson 89] Williamson, C., January 1989. Personal communication.

Processor Scheduling in Shared Memory Multiprocessors

John Zahorjan and Cathy McCann

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

September 1989

Abstract

Existing work indicates that the commonly used "single queue of runnable tasks" approach to scheduling shared memory multiprocessors can perform very poorly in a multiprogrammed environment. A more promising approach is the class of "two-level schedulers" in which the operating system deals solely with allocating processors to jobs while the individual jobs themselves perform task dispatching on those processors.

In this paper we compare two basic varieties of two-level schedulers. Those of the first type, static, make a single decision per job regarding the number of processors to allocate to it. Once the job has received its allocation, it is guaranteed to have exactly that number of processors available to it whenever it is active. Static schedulers are attractive to the system because of their low scheduling overhead and to the application because they provide an unchanging environment that simplifies decisions such as how many parallel tasks to fork.

The other class of two-level scheduler, dynamic, allows each job to acquire and release processors during its execution. By responding to the varying parallelism of the jobs, the dynamic scheduler promises higher processor utilizations at the cost of potentially greater scheduling overhead and more complicated application level task control policies.

Our results, obtained via simulation, highlight the tradeoffs between the static and dynamic approaches. We investigate how the choice of policy is affected by the cost of switching a processor from one job to another. We show that for a wide range of plausible overhead values, dynamic scheduling is superior to static scheduling. Within the class of static schedulers, we show that, in most cases, a simple "run to completion" scheme is preferable to a "co-scheduling" like round-robin approach. Finally, we investigate different techniques for tuning the allocation decisions required by the dynamic policies and quantify their effects on performance.

We believe our results are directly applicable to many existing shared memory parallel computers, which for the most part currently employ a simple "single queue of tasks" extension of basic sequential machine schedulers. We plan to validate our results in future work through implementation and experimentation on such a system.

This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, and CCR-8703049), the Naval Ocean Systems Center, the Washington Technology Center, and Digital Equipment Corporation (the External Research Program and the Systems Research Center).

Authors' addresses: Department of Computer Science and Engineering FR-35, University of Washington, Seattle, WA 98195; *zahorjan@cs.washington.edu, mccann@cs.washington.edu.*

1. Introduction

In this paper we consider alternative strategies for scheduling parallel jobs on multiprogrammed, shared memory parallel computers. We assume that at any point in time each job is composed of one or more ready tasks. The number of ready tasks, which we call the *parallelism* of the job, changes over time due to synchronization constraints of the computation. Each ready task can be executed in parallel with the others if there is a processor available to it. Thus, the rate of progress of an individual job is determined both by its parallelism and the number of processors that it has been allocated. The primary purpose of this paper is to examine a number of alternative strategies for allocating processors to jobs when many parallel jobs are competing for a fixed number of processors.

There are a number of architectural approaches to building parallel machines, and different scheduling strategies are suited to each. In this paper we are concerned with the simplest and to date most widespread architecture, the medium-scale, shared memory, uniform memory access (UMA) machine. By ''medium-scale'' we mean machines with a modest number of processors, say between 4 and 64. By ''shared memory'' we mean that any processor can reference any memory location, with the hardware providing the necessary control to transfer the data. By ''uniform memory access'' we mean that the time required to access a memory location does not depend on the identity of the processor making the access.

In practice, probably no machine fits the pure definitions given above. However, we intend our analysis to apply to machines such as the Encore, the Sequent [Lovett & Thakkar 1988], the Cray MPs, the IBM 3090 MPs, and the DEC Firefly [Thacker et al. 1988]. The most significant deviation of these machines from our assumptions is that they all have some sort of processor local memory, in particular, either a hardware managed cache or a user managed local store. Because some benefit may accrue from scheduling a job on a processor where it has run previously (since the local memory there may still contain information useful to that job), these local memories complicate scheduling decisions. While it may be possible to exploit this effect in tuning specific scheduling disciplines, for this class of parallel machine the more basic, unanswered questions about the broad divisions among processor allocation strategies are of primary importance.

We compare the performance of two fundamental approaches to processor allocation. In the first, *static allocation*, the number of processors available to each job is fixed during its entire execution. In the alternative approach, *dynamic allocation*, the number of processors allocated to a job may vary during execution in a way that reflects its time varying parallelism.

While at first it may seem that the performance of dynamic schedulers should dominate that of static schedulers, overhead is one reason that this may not be the case. Because dynamic schedulers tend to shift processors more frequently from one job to another, and because this ''context switch'' can be quite expensive, the overhead costs of dynamic scheduling can outweigh the benefits of reallocation.

Lower overhead cost is not the only factor favoring static schedulers, though. From the system's viewpoint, static schedulers are simpler to implement. From the job's viewpoint, knowing in advance the exact allocation of processors that will be available at all times during its execution may allow it to run more efficiently. For example, consider the job of performing a parallel matrix multiply. The structure of the job is given by the task graph shown in Figure 1. The sequential portion of the work, represented by task 0, determines the *granularity* of the parallel work, which in this case means the number of elements of the result matrix computed by each parallel task. Finer granularity, that is, more parallel tasks each of which performs a smaller fraction of the total work, results in shorter execution times (modulo the effects of task dispatching overhead) if there are sufficiently many processors available to run these tasks. On the other hand, consider execution time if too few processors are available to the job. Suppose that work requiring total time 1 is divided into nine equal pieces. If nine processors are available, the elapsed time of the computation is 1/9.

However, if only eight processors are available the elapsed time is doubled: eight of the nine tasks complete in time 1/9, then the final task is finished in additional time 1/9. This significant increase in elapsed time hurts not only the individual job but also the system as a whole, since during the second 1/9 time unit seven of the eight processors must either sit idle or suffer a context switch overhead. (While multiplexing the tasks on the available processors would eliminate this "end effect", suspending and resuming tasks is generally too expensive to make this a viable approach. Note that the cost of suspending a task may greatly exceed that of starting a new task, since in the former case considerable state information needs to be saved.)

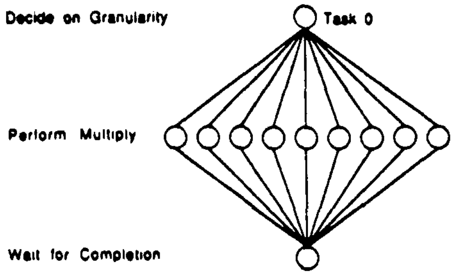


Figure 1 - Task Graph of Matrix Multiply

This sensitivity of job performance to processor allocation supports our interest in static schedulers. It also motivates a more basic assumption of this work, that both the operating system and the application are involved in making scheduling decisions. Under this "two-level" approach to scheduling, the operating system is responsible for partitioning the processors among the jobs. Each job uses the processors currently in its partition to execute some subset of its runnable tasks. An appropriate decision about which subset to run, as well as which task to suspend when a processor is pre-empted, is thus left to the individual applications, which are in the best position to understand the important synchronization relationships among their tasks. At the same time, the operating system retains its traditional scheduling role, that of managing the allocation of resources among competing jobs.

In contrast to two-level scheduling, existing schedulers for shared memory UMA parallel machines (e.g, Dynix [Lovett & Thakkar 1988] and Mach [Young et al. 1987]) typically maintain a single queue of runnable tasks with all processors in the system cycling among them in a round-robin fashion. This approach, which is a straightforward extension of scheduling in single processor systems, can perform very poorly on parallel jobs that synchronize [Ousterhout 1982, Zahorjan et al. 1988, 1989]. Our goal is to find scheduling strategies appropriate to parallel jobs.

Our interest in this work is both theoretical and practical. On the theoretic side, we would like to understand the circumstances under which each of static and dynamic scheduling is preferable to the other, and the magnitude of the performance differences exhibited. On the practical side, we are interested in policies that can be implemented. (In fact, the results of this work are guiding an implementation effort to modify the existing scheduler on our Sequent.) This means that the policies we consider are restricted to those that one might reasonably expect to be implementable, and that our models contain characteristics, such as overhead, that are important when scheduling real systems.

In the next section we define the system model and analysis approach that we have used, and in Section 3 detail the basic scheduling policies studied. In Section 4 we make a broad comparison of the performance of the two basic scheduling strategies, static and dynamic, with emphasis on determining the influence of processor allocation overhead on the choice between them. In Section 5 we examine the more promising approach, dynamic scheduling, in greater depth in an attempt to "tune" the scheduler for better performance. Finally, Section 6 presents a summary of our conclusions.

2. Model Definition and Analysis

Because we are interested in the effects of changes to some relatively low level details of scheduling policies, we have used simulation to obtain performance estimates. This has allowed us to make the comparisons envisioned at the outset of this study as well as to follow paths not originally foreseen. The simulator was implemented as a C++ program, the bulk of which is tailored to this particular application.

While in theory simulation allows almost any level of detail in the model, it is nonetheless advantageous to make the model as simple as possible. This typically results in faster simulation execution times (a matter of some concern), but more importantly aids in understanding the model results.

There are two components to our model, the representation of the hardware and the representation of the software. Our hardware model consists of P identical processors. In the examples shown later in this paper, we let $P=20$, a reasonable value for machines of the kind we address. Qualitative results for other numbers of processors are similar to those for $P=20$.

We assume that any ready task can obtain equivalent service from any processor. This implies a UMA memory organization, as mentioned previously. We do not explicitly model either main memory or I/O. While both can be significant factors in some parallel machines for some applications, in general the focus of parallel computations is on processor cycles. Thus, for this initial study we do not deal explicitly with these other factors.

We model processor allocation overhead, which we denote O, as a time delay between the first moment an eligible processor is available to satisfy an allocation request (either because of the generation of a new request or the change in status of some processor) and the time the processor actually becomes available to the job making the request. This delay represents a number of factors: the time required to run the scheduler code, the time required to save the context (if necessary) of the process running on the processor, the time required to load the new context, and the penalty associated with the expected low cache hit ratio during initial execution of that process. In our model there is no penalty associated with releasing a processor, since the time required to do so is either incorporated into the acquisition penalty just described (in the case that the processor is released to another job) or occurs when there is no immediate demand for the processor (in the case that the processor is released to a system pool of free processors).

We model the software component of the system as a Poisson arrival stream of parallel jobs. While our hardware model can be relatively simple, we decided that our model of these parallel jobs should be fairly detailed. Thus, in contrast to using some aggregate measure of parallel behavior (e.g., the fraction of sequential code [Amdahl 1967] or the average parallelism measure [Eager et al. 1989, Sevcik 1989]), we represent jobs explicitly by their task graphs. The motivation for this is that the comparison between static and dynamic scheduling policies obviously depends on the nature of the jobs to be processed. If we hope to obtain meaningful quantitative results about the situations in which one policy outperforms the other, it seems necessary to include the complex behavior of real jobs.

With this in mind, we have used three benchmark workload classes throughout this paper. Figure 2a shows the task graph structure of a parallel Mean Value Analysis [Reiser & Lavenberg 1980, Almquist et al. 1989] solution package for product form queueing networks containing two classes of N customers each. (In Figure 2a N equals 10.) We refer to such a job as an MVA(N,N) job. We expect all the tasks in this application to have identical mean execution times, since they perform an identical amount of work. Thus, if a processor were always available for use when a task became runnable, the parallelism of the job would step through the sequence 1, 2, 3, ..., N, $N+1$, N, ..., 3, 2, 1 during its execution. This represents a very gradual but, for larger N, significant change in parallelism over the life of the job. We expect this characteristic to favor dynamic scheduling, can adapt efficiently to the slowly changing processor demand.

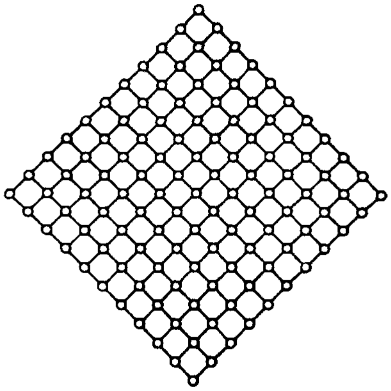


Figure 2a - MVA(10,10) Job

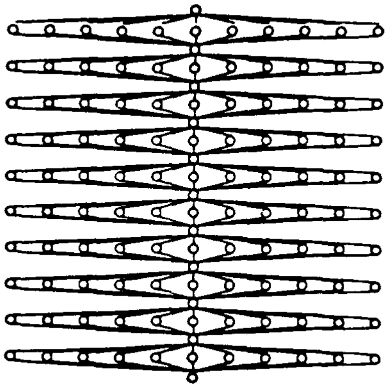


Figure 2b - Fork-Join(11) Job

We let the mean task service time of MVA(N,N) jobs be the unit of time in the model, i.e., the mean task service time is equal to 1.0. Although we expect each task to take time nearly equal to 1.0, in an actual system there will be some variation in task execution times caused by, for instance, slightly different cache hit ratios and contention for access to shared memory. To reflect this in the model, we choose task service times independently and uniformly from 0.95 to 1.05.

In our examples we use two specific members of the MVA class of jobs. The first, MVA(2,2), has a maximum parallelism of 3 and a mean total processor time of 9 The other, MVA(10,10), has a maximum parallelism of 11 and mean total processor time of 121. Clearly, the smaller jobs have more nearly constant parallelism and so we expect static scheduling to be more effective on them than on the larger ones.

The second benchmark application is the canonical Fork-Join (FJ) job, whose task graph structure is shown in Figure 2b. These jobs cycle between sequential phases and phases with K runnable tasks. Fork-Join jobs arise naturally in jobs that exhibit "data parallelism", meaning those that apply the same computation to a number of different data points. Many scientific computations over grids fall into this class, as does the matrix multiply example of Figure 1. The repeated fork-join cycle in the job reflects the often iterative nature of these computations.

In contrast to MVA jobs, FJ jobs exhibit abrupt changes in their parallelism during execution. To help isolate the effects of job structure on the relative performance of differing scheduling disciplines, we parameterize the FJ jobs in such a way that some of their important measures match those of the MVA jobs. In particular, we consider two different instances of the Fork-Join class. In each, task services times are chosen as for the MVA class of jobs (i.e., independently and uniformly from 0.95 to 1.05). The first instance of the FJ job class, denoted FJ(3), forks into 3 tasks after each sequential phase and repeats the fork-join cycle 2 times. This leads to a job with maximum parallelism 3 and mean total processor time 9, which match the corresponding statistics for the MVA(2,2) jobs. For the second instance, FJ(11), each sequential phase is followed by 11 parallel tasks and the cycle is repeated 10 times. This gives maximum parallelism 11 and mean total processor time 121, identical to those measures for the MVA(10,10) jobs.

Because the FJ jobs vary their parallelism quite rapidly, we expect that they form a kind of stress test for dynamic scheduling, whose natural tendency is to move processors from job to job in an attempt to track these changes in parallelism. The potentially high rate of processor reallocation that results can lead to significant context switch overhead, resulting in poor performance.

The final job class, called Variable-Fork-Join (VFJ), has the same basic structure as the fork join jobs, that is, repeated cycles of sequential followed by parallel phases. However, for this class we assume that while the total work to be done during the parallel phase is fixed, the number of tasks forked to perform the work is determined at the end of the immediately preceding sequential phase. This allows the job, for instance, to divide the total work into a number of pieces that suits the number of processors available to it. In the case of static scheduling, a VFJ job splits into as many tasks as it has processors. In the case of dynamic scheduling, the job forks into a number of tasks equal to the number of processors already allocated to it plus the number it can begin acquiring immediately.

We do not charge any overhead in partitioning the total parallel phase work among the forked tasks in the VFJ jobs. Thus, the mean execution time of each resulting task is equal to the total time of the parallel phase divided by the number of tasks generated, and the variation among task execution times is small. (We once again assume a uniform distribution from 5% below to 5% above the mean task execution time.) In general, such a situation cannot be achieved in practice for two reasons: the inability to divide the load equally among the tasks and contention for the queue of descriptors describing the work remaining to be accomplished [Kruskal & Weiss 1985, Polychronopoulos & Kuck 1987]. The former results in mean task times that may differ significantly from one task to the next while the latter inflates the total processor busy time required to accomplish the work. Both these overheads are greater for larger numbers of forked tasks. Thus, our results for this workload tend to be a little optimistic.

The particular instances of the VFJ job class we study are the analogs of the Fork-Join jobs described earlier. The smaller jobs, denoted VFJ(3), have two fork-join cycles requiring a total of 3 units of work during each parallel phase. The larger jobs, denoted VJF(11), have twelve fork-join cycles with total work 11 during each parallel phase.

Workload	Minimum Parallelism	Maximum Parallelism	Average Parallelism	Total Processor Time	Minimum Elapsed Execution Time
MVA(2,2)	1	3	1.8	9	5
FJ(3)	1	3	1.8	9	5
VFJ(3)	1	P	$3P / P+2$	9	$3 + 6/P$
MVA(10,10)	1	11	5.8	121	21
FJ(11)	1	11	5.8	121	21
VFJ(11)	1	P	$11P / (P+10)$	121	$11 + 110/P$

Table 1 - Summary of Workload Characteristics

Table 1 gives some simple measures of the workloads used in this study. "Total Service Time" is the sum of the mean task times. "Minimum Parallelism", "Maximum Parallelism", and "Average Parallelism" are those measures under the assumptions that context switch overhead has zero cost, that all task execution times are deterministic, and that all P processors are available to the job. (For the MVA and FJ jobs we have assumed that P is greater than their maximum parallelisms, which is true of all the examples we present.) "Minimum Elapsed Execution Time" is the elapsed time required to execute a job under these same assumptions.

Because the magnitude of the processor allocation overhead has a major effect on the performance of scheduling policies (dynamic has an advantage when overhead is very small, static when it is large), it is important to set this parameter of the model appropriately. Unfortunately, determining a reasonable value is quite difficult. Since the overhead cost is given as the ratio of the absolute processor allocation time to absolute average task execution time, it depends on characteristics of both the parallel machine considered and the individual application. Thus, as overhead will vary from system to system, and even from job to job on the same system, all our experiments are performed over a range of overhead values.

In an attempt to identify a reasonable range of values, we have measured overhead and task times on our Sequent Symmetry. The time to switch a processor from one job to another is roughly 750 µsec. The

MVA(10,10) task time is about 75 μsec. per service center in the queueing network being solved by MVA. This translates to an overhead range of roughly 0.0 to no more than 2.0 time units for reasonable assumptions about the size of the queueing network. (Remember that the time unit in the model is the execution time of a single task.) We also measured the matrix multiply rate on the Sequent, as a representative application of the FJ job structure. Each element of the matrix resulting from multiplying together two 100x100 matrices requires about 1525 μsec. Dividing the total work equally among 20 processors (the number on our Sequent) results in tasks times of about 762,500 μsec., or overhead times of nearly 0.0

Based on these measurements we have chosen to examine an overhead range of 0.0 to 2.0. We believe that for most computations the range 0.0 to less than 1.0 is of most interest.

3. The Basic Policies

In this section we define more fully the scheduling disciplines examined in our initial comparison of the static and dynamic approaches. As mentioned previously, we assume a "two-level scheduling" mechanism where the processor allocation function is performed by the operating system and the task dispatching function is performed by each job.

We discuss here the static and dynamic processor allocation policies studied. We do not attempt to specify precisely how task dispatching is performed. (Graham [1966] has shown that the order in which tasks are dispatched in our model can affect elapsed execution time by no more than a factor of 2–1/*P* when the job runs with *P* processors.) However, we make the assumption that a task, once started, is run to completion[1]. This is motivated by consideration of overhead, which renders round-robin execution of all runnable tasks (the natural alternative) impractical for many parallel applications. It also implies that tasks block (that is, relinquish their processor to another task of the same job) rather than spin at synchronization points when there are waiting runnable tasks. Fast thread packages [Bershad et al. 1988, Tucker & Gupta 1989, Birrell 1989] make such a blocking strategy practical, and previous modelling work has shown that the expected performance of spinning and blocking in this case is quite similar [Zahorjan et al. 1989].

3.1. The Static Policies

We consider two specific implementations of static scheduling. In both cases the number of processors allocated to a job is fixed by a single decision made at the onset of the job's execution. However, the two policies differ in their manner of controlling the set of runnable jobs.

Under the first static policy, called Run-To-Completion (RTC), once initiated each job is allowed to continue execution uninterrupted until termination. When a job arrives to the system, if there are free processors some number (potentially all) of them are allocated to it and it begins execution. If no processors are available, the jobs waits for some currently executing job to finish. At that point, the processors released by the completing job are allocated among all waiting jobs. (The method for deciding how many processors to allocate each arriving job is described shortly.)

The alternative static policy, Round-Robin, is based on the notion off co-scheduling [Ousterhout 1982]. Round-Robin cycles all of the processors in a round-robin fashion among all jobs in the system. That is, at the end of each scheduling quantum all *P* processors in the system are pre-empted from the set of currently running jobs and given to a different set of jobs. Note that this is a static policy since a particular job is allocated the same number of processors during each quantum that it runs.

---

[1] Round-robin execution can be incorporated into our model in an approximate way by breaking each task into a sequence of smaller tasks. The task dispatcher will then tend to circulate the processors among all the original tasks in a fair manner.

To complete the definitions of RTC and Round-Robin we must specify how each decides upon the number of processors to allocate an arriving job. In the absence of *a priori* information about the job, this decision must be made in a job independent way (although the decision could depend on recent system measurements, such as the average processor utilization or the current number of runnable jobs). As an example, RTC might simply assign each job some fixed number of processors, or perhaps all processors that are available when the job arrives, while Round-Robin might assign all processors to each job.

Rather than pursue this approach we have assumed that each job provides an accurate characterization of itself to the static scheduler when the job arrives. (One advantage of the dynamic scheduler, as we will see, is that it has no need for this *a priori* information.) In practice, this is somewhat unrealistic both because users may not know the characteristics (e.g., if job behavior is data dependent) and because the system cannot trust users to truthfully represent their workload [Coffman & Kleinrock 1968]. However, by giving the benefit of the doubt to the static policies we strengthen our confidence that dynamic scheduling will in practice be preferable in those circumstances where it dominates in our model.

The characterizations of a job j used by the static schedulers are its total execution time, denoted $T_j(1)$, and its speedup curve. The speedup of job j when allocated p_j processors is denoted $S_j(p_j)$, and is given by the ratio of $T_j(1)$ to $T_j(p_j)$, its elapsed execution time when run on p_j processors. Intuitively, the speedup curve represents the incremental benefit of allocating additional processors to a job.

From these input quantities we can derive two other measures of interest, the average and maximum parallelisms. Average parallelism, denoted A_j, is defined as the average number of processors the job would keep busy if it had an unbounded number of processors available to it. With this definition, A_j is given by $S_j(\infty)$. Maximal parallelism, denoted M_j, is defined as the smallest number of processors for which the speedup curve achieves its maximum value.

3.1.1. The RTC Policy

Given the quantities supplied by the job upon arrival, the RTC scheduler operates as follows:

RTC

Upon job arrival:

1. If there are idle processors, the job is allocated the lesser of the number of idle processors and the job's maximum parallelism, M_j.
2. Otherwise, the job waits until processors become available.

Upon job completion:

3. Each of the processors released by the completed job is assigned in turn to that waiting job j whose current assignment of processors p_j is less than M_j and for which the expected improvement in elapsed execution time, $T_j(p_j) - T_j(p_j+1)$, is greatest. When all released processors have been assigned, the processors are actually allocated to the jobs, which begin execution (after a processor allocation overhead time).

In Appendix A we show that the greedy allocation of Step 3 provides minimal average response time over the current set of jobs (i.e, ignoring any future arrivals or departures by jobs already in service, which cannot be predicted by the operating system in practice). By defining $T_j(0)$ to be very large, Step 3 implies that any waiting job not yet assigned a processor will be assigned one before any other job is assigned more than one.

3.1.2. The Round-Robin Policy

To fully explain the strategy followed by the Round-Robin allocator we must first introduce a bit more notation. Assume that each running job k has been assigned p_k processors and has been placed in a scheduling "slot". All jobs in any one slot run simultaneously. Processors are assigned to slots on a quantum-driven basis, with all P processors moving from slot to slot at each quantum expiry. Let there be a total of N slots and for each slot n let $F_n \equiv P - \sum_{k \in n} p_k$ be the total number of unallocated processors in that slot. The basic problem confronting the Round-Robin scheduler when a job arrives is whether to allocate it the F_n free processors in some existing slot n or to create a new slot and allocate it some larger number (up to P) of processors. Creating a new slot allows the job to progress more quickly when its slot is scheduled, but, by increasing the total number of slots, reduces the rate at which each slot is scheduled.

Unlike the RTC scheduler, whose objective function involves only the jobs currently being scheduled, Round-Robin makes a global decision in assigning the new job to either an existing or a new slot since the decision may affect the response times of all jobs in the system. Round-Robin's objective is to maximize the total rate of delivery of useful computing in the system.

Let $C \equiv \sum_{\text{all jobs } k} S_k(p_k)$ represent the current effective computing rate of the system. Ignoring the cost of context switching, $\frac{C}{N}$ gives the average number of processors usefully busy under the existing allocation. The Round-Robin policy then is:

Round-Robin

Upon arrival of job j:

1. Let slot max be the one with the most unallocated processors. Then if $F_{\max}$ is greater than or equal to A_j, the job's average parallelism, the job is allocated to slot max and is given A_j processors.
2. Otherwise, if

$$\frac{S_j(F_{\max}) + C}{N} \geq \frac{S_j(Min(P, A_j)) + C}{N+1}$$

   the job is allocated $F_{\max}$ processors in slot max.
3. If allocating to a new slot, $N+1$, allocate the job $Min(P, A_j)$ processors.
4. If job j is allocated to the slot currently in execution and that slot has processors not used at the moment (see Step 6), begin executing j as long as the remaining time in the quantum exceeds the context switch overhead time O.

Upon the completion of a quantum for slot n:

5. If there have been any job completions during the quantum, calculate the new number of unallocated processors F_n in this slot. Scan the list of slots sequentially. If the number of allocated processors in a scanned slot is less than F_n combine that slot with slot n, update F_n, and stop the scan. (It is not possible for two existing slots to be combined with n or those slots would already have been combined with each other.)

When scheduling slot n:

6. Allocate the F_n unused processors to jobs in the immediately succeeding slot, giving multiple processors to a single job in preference to fewer processors to many jobs. (The number of allocated processors in the next slot must be greater than the F_n or the two slots would have been combined by Step 5.)

The upper bound allocation imposed in Steps 1 and 3 is motivated by possible future arrivals. Since the current job achieves nearly optimal performance with allocation A_j [Eager et al. 1988] any idle processors in excess of this number "are saved for future arrivals." We did not have to take this conservative approach under RTC because, as we shall see, RTC naturally reduces the allocation of processors below the maximum parallelism of the jobs whenever there is a significant load on the system.

We note that fragmentation of allocation among the slots is a major problem with Round-Robin. Steps 4, 5, and 6 are all concerned with this effect. Our simulations indicate that the performance of Round-Robin is significantly reduced if processors are allowed to go unused. The magnitude of this degradation was surprising; we had originally formulated a simpler version of Round-Robin anticipating that it would perform nearly as well as this more complicated scheme. However, the performance of the simpler approach was unacceptable, being far inferior to both RTC and Dynamic under all parameterizations of the model.

3.1.3. Qualitative Comparison of RTC and Round-Robin

There are a number of qualitative distinctions between the Round-Robin and RTC approaches. Round-Robin affords greater flexibility in making processor allocation decisions than RTC (since the sum of all currently decided allocations can exceed the number of processors on the system), a factor that favors Round-Robin. However, Round-Robin incurs more overhead because of the cost of quantum pre-emptions. This favors RTC. The round-robin aspect of Round-Robin is a better approximation to Shortest-Job-First than RTC, a desirable attribute in minimizing response times, although the benefit of this is less pronounced in multiprocessor systems than in sequential machines [Sauer & Chandy 1979]. Whereas under Round-Robin the delay between job arrival and the start of its execution is relatively short compared with that under RTC, the subsequent delay until execution completes is relatively long.

While the qualitative differences between the two static policies are quite clear, the quantitative differences in their performance are not. This is one of the questions addressed in Section 4.

3.2. The Dynamic Policy

In contrast to the two static schedulers, the Dynamic scheduler allows jobs to gain and release processors during their executions. In particular, the job's task dispatcher can request additional processors whenever they are needed, and the operating system processor allocator will attempt to meet these requests. The task dispatcher can unilaterally release processors (to the operating system) at any time.

The basic operation of the policy is quite simple. To define it fully, we must specify the actions taken by both the processor allocator in assigning processors to jobs and the decision procedure followed by the task dispatchers in requesting and releasing processors:

Dynamic Processor Allocator

When a job requests one or more processors (including job arrival):

1. If there are idle processors, use them to satisfy the request.
2. Otherwise, if the job making the request is a new arrival, allocate it a single processor by taking one away from any job currently allocated two or more.
3. If any portion of the request cannot be satisfied, it remains outstanding until either a processor becomes available for it or the task dispatcher rescinds it (e.g., if the job's parallelism decreases in the interim).

Upon release of one or more processors (including job departure):

4. Scan the current queue of unsatisfied requests for processors. Assign a single processor to each job in the list that currently has no processors (i.e., to all waiting new arrivals). After these jobs have

been assigned a processor, scan the list again, allocating the rest of the processors on a FCFS basis.

Steps 2 and 4 assure that newly arriving jobs receive a processor relatively promptly. We found this to be important to performance, an effect also noted by others [Majumdar & Eager 1988].

The application's task dispatcher is run after the completion of each task. While each job is free to perform task dispatching (including making requests for and releasing processors) according to any policy it chooses, for the initial comparisons of dynamic and static scheduling in Section 4 we assume that all jobs use the following procedure:

Dynamic Task Dispatcher

During job execution, at instants that a processor finishes a job j task:

5. If there is a newly arrived job that has not yet received a processor and job j holds two or more processors, the task dispatcher releases its processor.
6. Otherwise, let q_j be the number of job j ready tasks waiting for processors. If $q_j = 1$ the task dispatcher simply dequeues the task and starts its execution.
7. If $q_j > 1$ the task dispatcher advertises that job j wants $q_j - 1$ additional processors assigned to it and then begins executing the first task in the queue.
8. Finally, if $q_j = 0$, the task dispatcher examines the information posted by other jobs to see if any of them want additional processors. If so, it releases the processor. On the other hand, if $q_j = 0$ but no other job currently wants additional processors, the dispatcher sits in a loop examining the ready queue for j and the processor request information posted by other jobs. If a task arrives to j's ready queue before the any other jobs make processor requests, the task can begin executing without a context switch penalty. (Otherwise, the processor is released.)

We note that the Dynamic policy as described is very "eager", meaning it gives up and acquires processors as quickly as it can. Under some circumstances this can lead to significant context switch overhead. In Section 5 we examine modifications to the Dynamic policy that reduce this overhead.

There is one additional consideration not addressed by the Dynamic policy as we have described it. In an actual system one might expect users to take simple countermeasures [Coffman & Kleinrock 1968] to the Dynamic policy by modifying the behavior of the task dispatcher (which is part of the application). For example, the task dispatcher might ignore Step 8 (by never releasing processors) or might inflate the size of q_j in Step 7. Partially in anticipation of this problem the policy implemented in our simulator actually incorporates a multi-level feedback priority mechanism. A job begins running at highest priority. The total number of processor-seconds accumulated by the job is monitored, and as it crosses a set of parameterizable thresholds the job drops in priority. A job at a lower priority may have its processors pre-empted for allocation to a higher priority job. This has two effects. The first is to reduce the advantage of the countermeasures listed above, since holding onto excess processors causes a job to drop in priority more quickly. The second effect is to provide a better approximation to Shortest-Job-First (under the common assumption that a job that has already run for a long time will most likely continue to run for a long time).

We have not used this feedback mechanism in the examples shown in the next section, as it did not lead to changes in average response time. In those examples involving only one workload class this might be attributed to the fact that this modification renders Dynamic *less* like Shortest-Job-First. (Since jobs have nearly deterministic service requirements, a newly arriving job must have a greater remaining service requirement than a job already in execution.) The failure to improve response times in the multiple class examples is probably due to the same basic effect. If our job classes showed much more extreme differences in execution time requirements, it is likely that some benefit would be realized.

4. Comparison of the Policies

In this section we compare the performance of the three schedulers, RTC, Round-Robin, and Dynamic. All results were obtained by simulation. Each simulation was run for 15500 time units and statistics from the initial 500 time unit interval were thrown out to reduce the effect of initial start-up. This run length produced 95% confidence intervals that were typically much less than one percent of the mean for the measures we examined.

All simulations were parameterized for a system with $P=20$ processors. The task service times were chosen uniformly from 0.95 to 1.05. The quantum length for Round-Robin was chosen to so that the fraction of time spent in overhead due to quantum expirations, $\frac{O}{O+\text{quantum length}}$, was 2.5%. This value, which was arrived at through experimentation, represents a compromise between low overhead and reasonably short quanta, and typically results in nearly minimal job response times.

In each experiment we varied the context switch overhead O from 0.0 to 2.0 time units. (Remember that a time unit is defined to be the mean task execution time.) We note that there are two ways to interpret the magnitude of the context switch time. In the first, one assumes that the workload is given, thus fixing the absolute task service times, and our results indicate how fast context switching must be made (by a combination of the hardware and kernel designers) for dynamic scheduling to be preferable to static scheduling. The second interpretation is to assume that the system, and thus the context switch time, is fixed and that we are deciding on the granularity of the job. Under this interpretation, an overhead of 0.0 corresponds to very coarse granularity and overhead 2.0 corresponds to very fine granularity. Then, if the experiments indicate that dynamic scheduling is preferable to static for, say, $O<1.5$, a job should either chose a granularity that results in task times of at least 2/3 the actual context switch time or should take a countermeasure to the scheduler that causes it to allow the job an effectively constant allocation.

For simplicity in what follows we assume that the overhead costs of all disciplines are equal. In practice, the differing complexities of the various schedulers would allow some to make decisions more quickly than others. This can be taken into account in an informal way by comparing the performance of the static schedulers at overhead costs lower than those taken for the dynamic scheduler. However, we will see that the performance of the static schedulers are relatively insensitive to overhead costs. Thus, even if the static schedulers resulted in significantly lower overhead costs than those under the Dynamic policy, this would not greatly change the conclusions we reach by assuming that the costs are equal.

The performance metric by which the schedulers are compared is mean job response time. However, we have also extracted a number of other measurements from the simulations to help in drawing conclusions. Among these are the mean fraction of time each processor spends in overhead, the mean fraction of time each spends allocated to a job but unused, the mean number of processors allocated to each job, the mean time between job arrival and the first acquisition of processors, and the mean time from start of execution to job completion.

We compare the performance of the schedulers over different workload types and volumes. We set workload volumes by adjusting job arrival rates to achieve a specified useful utilization of each processor. If U is our target useful utilization, $T(1)$ is the average total execution requirement of the jobs in the workload, and there are P processors in the system, the arrival rate λ is set by $U=(\lambda\, T(1))/P$. For example, in a system with 20 processors, to achieve a target utilization of 75% with a workload consisting entirely of FJ(11) jobs, the arrival rate is set to (20)(0.75)/121.

Because of the complexity of the system we are studying, we first examine the results of a set of single-class experiments in an attempt to uncover the relationships among the schedulers. We follow this with an examination of multiple-class models to see how the tradeoffs among the schedulers are affected.

4.1. Single Class Comparisons

In Figures 3a-3f we compare the schedulers on our six workloads. In each case, a Poisson arrival stream of a single workload is generated at a rate that produces an average useful processor utilization of 50%. Each graph shows average response time as a function of context switch overhead.

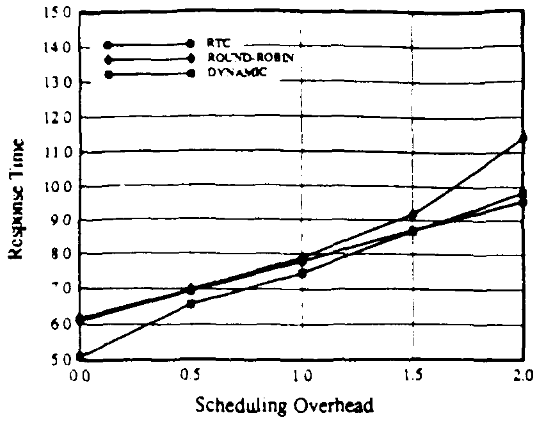


Figure 3a - Simulation of MVA (2,2)

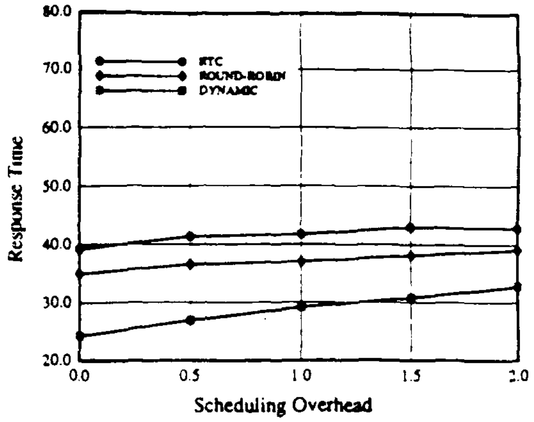


Figure 3d - Simulation of MVA (10,10)

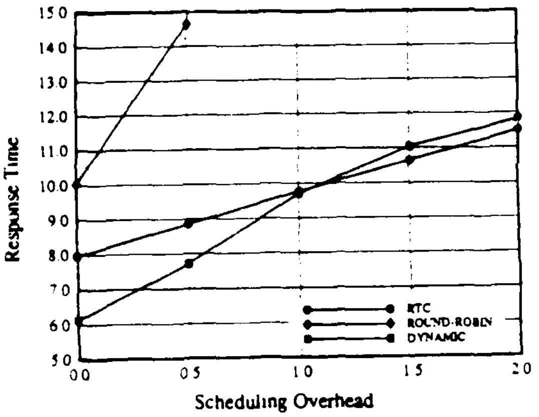


Figure 3b - Simulation of FJ (3)

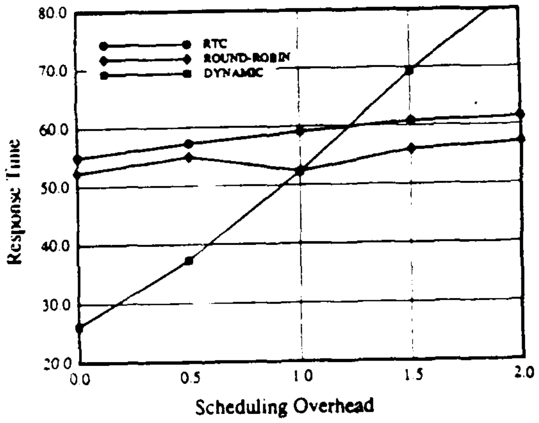


Figure 3e - Simulation of FJ (11)

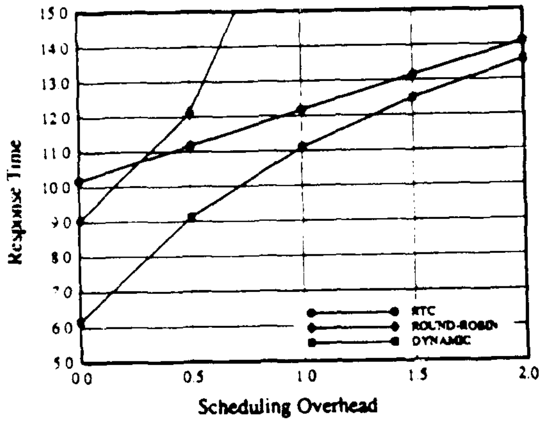


Figure 3c - Simulation of VFJ (3)

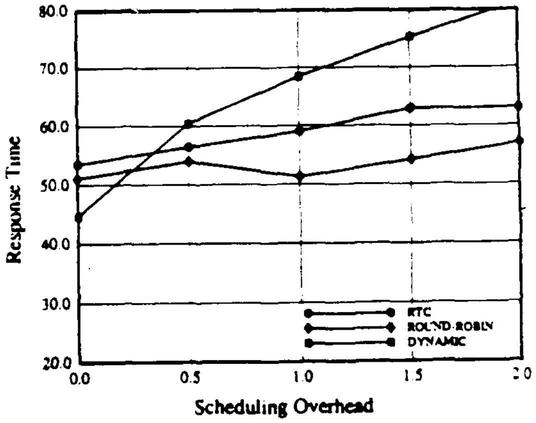


Figure 3f - Simulation of VFJ (11)

Based on these results, as well as those of other experiments not shown here, we reach the following conclusions:

Static vs. Dynamic Scheduling:

1. *Independently of workload (and overall system load), dynamic scheduling is best for small context switch overheads.*

   This is as we expected. When context switching is cheap, Dynamic allows maximum utilization of the processors by adapting allocation to instantaneous parallelism. However, when context switching is expensive, the cost of exploiting changes in parallelism sometimes outweighs the benefits.

2. *The advantage of Dynamic at low context switch costs increases with larger and more rapid changes in the parallelism exhibited by a workload.*

   In Figure 3, comparing the results of MVA(2,2) with MVA(10,10) and FJ(3) with FJ(11) we see that the advantage of Dynamic over the static schedulers is nominal for the small jobs but significant for the larger ones. (The behavior of the VFJ jobs is explained in point 6 below.) The reason for this is evident: the smaller jobs exhibit little change in parallelism, so there is not much difference between the Dynamic and static policies. On the other hand, the large jobs do change their parallelism considerably, and Dynamic is able to make better use of the processors over the lives of the jobs than the static policies.

   Figure 3 also points out another effect: the threshold beyond which a static scheduler outperforms Dynamic decreases with the rapidity of fluctuation in parallelism. This is seem by comparing the FJ(11) with the MVA(10,10) jobs. Experiments with effective loads higher (75%) and lower (25%) than those of the experiments shown in Figure 3 show very similar behavior. In fact, the values of the thresholds in the experiments for each workload type are nearly independent of the system load. (See, for example, Figure 4.)

3. *The relative advantage offered by Dynamic increases with increasing load: the advantage offered at low context switch costs is magnified and the disadvantage at high context switch costs diminished.*

   This effect is illustrated in Figure 4, which shows the performance of RTC, Round-Robin, and Dynamic on the FJ(11) workload for system loads of 25%, 50%, and 75%. (Round-Robin produces response times that are off the scale for the 75% load.)

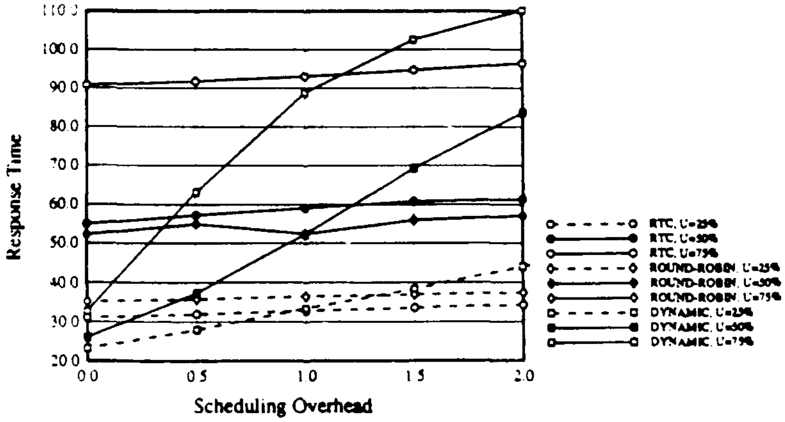


Figure 4 - FJ(11) at 25%, 50%, and 75% Useful Utilization

At high system loads there are large penalties associated with even small reductions in the average number of processors doing useful work. Both the static and Dynamic policies lose processing power to context switch overhead. The static policies also suffer an additional problem, fragmentation in their allocations caused either by a job holding processors it does not currently need or needing more processors than it has been allocated at a time when there are idle processors in the system. Dynamic does not experience this fragmentation. Thus, when context switch times are sufficiently small, Dynamic has a significant advantage over the static alternatives at high load.

Conversely, when the system load is fairly light, making effective use of all the processors is not as important. Thus, the static policies can afford to make large allocations. Under these conditions the Dynamic policy suffers because a job receives no more processors than it would under the static schedulers but experiences multiple context switches in acquiring them.

Round-Robin vs. RTC:

4. *On the whole, RTC is preferable to Round-Robin for general purpose use.*

Round-Robin yields response times lower than those of RTC only at medium loads, and even there the advantages are quite modest. (Figure 3d is the maximum advantage we have observed.) On the other hand, Round-Robin is unstable, producing very large response times in environments handled easily by RTC. (See, for example, Figures 3b and 4.)

This behavior is explained by the decision procedures used by the two policies for allocating processors. In general, a static policy should assign many processors to a job when the system load is low and only a few processors when load is high [Sevcik 1989]. Under RTC, the number of processors allocated to an arriving job j is constrained by

$$\text{Allocation}_j^{\text{RTC}} \le Min\,(M_j,\ \#\ \text{Idle (or Released) Processors})$$

Thus, RTC has the proper behavior. When load is light there are many idle processors, and RTC makes a maximal allocation. When load is heavy there are few idle processors and RTC reduces the allocation to each job.

On the other hand, Round-Robin does not have this property. Its constraint on allocation is simply

$$\text{Allocation}_j^{\text{Round-Robin}} \le A_j$$

Thus, whenever the job is allocated to a new scheduling slot it is given A_j processors, independently of the system load. While jobs are discouraged from creating new slots by Round-Robin's allocation policy, this must happen periodically. When it does, roughly $\left\lfloor \frac{P}{A_j} \right\rfloor$ arrivals are each allocated A_j processors. The next single arrival may be allocated the remaining processors in the slot, and the arrival following that triggers a new cycle of this behavior. Thus, when P is much larger than A_j, there is little reduction in average allocation in response to load.

The difference in allocations made by RTC and Round-Robin is evident in the allocations each makes to FJ(11) jobs when context switch cost is set to 1.0: RTC assigns an average of 9.19, 5.51, and 2.72 processors per job at system loads of 25%, 50%, and 75% respectively, whereas in the same circumstances Round-Robin assigns 5.76, 5.41, and 4.98 processors per job. We see that at low loads RTC assigns larger allocations than Round-Robin, resulting in slightly better performance[1]. At medium loads the allocations are nearly equal, and for this overhead value, so is performance. At high loads, however, Round-Robin's allocation of nearly 5 processors per job means that each allocated processor is busy only about 60% of the time. In contrast, under RTC allocated processors are usefully busy more than 80% of the time. (The same effect accounts for Round-Robin's poor performance with small jobs at medium load.)

We have been unable to find an alternative allocation strategy for Round-Robin that modifies this behavior qualitatively and is still in the philosophy of this approach, that is, and still allows the sum of the allocations to exceed the total number of processors in the system by an arbitrary amount. Thus, we conclude that the comparisons made in this section represent fundamental properties of the philosophies of the RTC and Round-Robin approaches in this environment.

---

[1] It is precisely because RTC adapts to load that it can afford a upper allocation bound of M_j whereas Round-Robin must adopt the more conservative bound of A_j.

5. *RTC shows surprising resilience to high loads.*

   We had expected that RTC would perform poorly as load increased because of the inefficiencies of a job holding a fixed number of processors even during periods when it cannot make use of them all. However, the tendency of RTC is to reduce allocations when there are many jobs in the system, so that when system performance becomes bad the average allocation is decreased to a single processor, eliminating any inefficiency.

   Figure 4 illustrates our observations about RTC. Here we see that response time under RTC increases by a factor of about 1.5 in going from system load of 50% to 75% useful utilization. In contrast, the wait time in an M/M/1 queue doubles. Note also that the performance of RTC is nearly independent of overhead cost regardless of system load.

FJ vs. VFJ Job Structure:

6. *In general, but particularly under the Dynamic scheduler, the FJ workload performs as well or better than the corresponding VFJ alternative.*

   Under the static schedulers, FJ and VFJ perform comparably when the system load is light to medium, but when system load is heavy the FJ structure is preferable, especially for small jobs. Under the Dynamic policy, FJ and VFJ perform comparably for small jobs, but FJ is considerably preferable for the large ones, especially at medium to heavy loads.

   When FJ is preferable to VFJ it is because the ability of the VFJ jobs to monopolize all the free processors causes other jobs to make do with only a few. In the case of the Dynamic scheduler, when one VFJ job holds many processors, other VFJ jobs arriving at their parallel phases will decide to partition their work into only a few tasks each (since there are only a few processors currently available to run them). Once this partitioning is performed the job is committed to it until all work in that phase is completed.

   In contrast, an FJ job splits into a constant number of pieces regardless of the current availability of processors. This means that it can take advantage of processors released by other jobs after the fork is performed. It is this distinction that gives FJ an advantage over VFJ.

   The same argument applies under the static policies at heavy load. When a small VFJ job arrives to the system at a time when there are many available processors, it is allocated all of them. Subsequent arrivals then find very few (if any) processors, and so are given small allocations. This results in a very unequal partitioning of the processors among the jobs, something that leads to low processor efficiencies and so poor performance. In contrast, an FJ job finding many free processors upon arrival takes only a limited number of them. Thus, the processor allocation is more equal and the overall processor efficiency is higher, resulting in better performance.

   We note that in practice this comparison most likely favors FJ even more strongly than it does in our model because of the optimistic assumptions we have made in defining the VJF workload. (See Section 2.)

4.2. Multiple Class Comparisons

We now turn our attention to the multiple class environment. We once again ran a set of experiments in which the system load was varied from 25% to 75% useful utilization. In each experiment, all six workload classes were given the same arrival rate. Thus, to achieve useful utilization of U we set the arrival rate λ of each class by $U = \lambda \frac{9+9+9+121+121+121}{20}$.

Examination of the results of these multiple class experiments and comparisons with the single class results lead us to the following conclusions:

1. *In terms of average response time, Dynamic scheduling dominates static scheduling for almost all overhead values considered.*

   Figure 5 shows response time averaged over all six workload types against context switch overhead for arrival rates yielding 25%, 50%, and 75% useful utilization. (The results for Round-Robin at 75% load are off the scale.) We see that Dynamic outperforms the static schedulers except for small differences at large overheads under light and medium loads. Keeping in mind that overheads are much more likely small than large, Dynamic seems the clear choice.

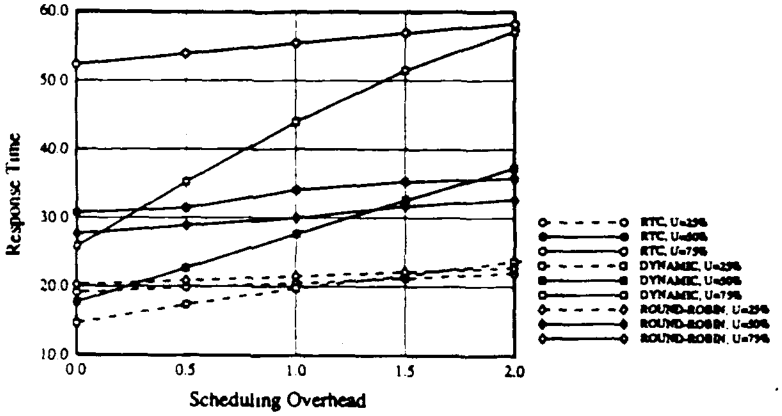


Figure 5 - Overall Response Times

We note also that the gain in advantage realized by Dynamic in moving from the single class to the multiple class environment results from a small decrease in response times under Dynamic and a small increase under the static schedulers. This is illustrated by Table 2, which gives individual class response times in a multiple class system with overhead cost equal to 1.0. (The large decreases in response times for FJ(3) and VJF(3) under Round-Robin result from their extremely poor behavior in the single class environment.)

	Single Class			Multiple Class		
	RTC	**R-R**	**Dyn**	**RTC**	**R-R**	**Dyn**
MVA(2,2)	7.80	7.80	7.36	10.39	11.34	7.53
FJ(3)	9.76	39.60	9.34	11.56	14.00	9.24
VFJ(3)	12.18	19.00	11.10	11.10	12.51	9.32
MVA(10,10)	41.71	37.03	29.16	48.81	40.48	28.33
FJ(11)	59.08	52.32	51.28	59.34	49.15	39.37
VFJ(11)	59.02	51.24	68.62	60.49	49.97	67.24

Table 2 - Single vs. Multiple Class Environments at 50% Load

2. *Dynamic performs increasingly well relative to the static policies as system load increases.*

   We note from Figure 5 that *benefit* of Dynamic at low overhead costs increases with increasing loads and that the *penalty* of dynamic at high overhead costs decreases to a negligible amount. Correspondingly, the threshold in overhead cost below which Dynamic is superior to RTC increases with increasing system load. The same trends also exist in terms of the performance of each individual class.

3. *There is no apparent benefit to the round-robin aspect of the Round-Robin policy.*

   The relationship of Round-Robin to RTC in the results of Figure 5 is identical to that evidenced in the single class experiments. Thus, the anticipated benefit of Round-Robin's better approximation of Shortest-Job-First did not materialize.

4.3. Summary of Comparisons

Looking over all the results of this section, it is clear that:

A. RTC is the most robust policy with respect to context switch overhead (and thus with respect to application granularity).

B. While Round-Robin sometimes performs better than RTC, the benefit in these cases is small and does not justify risking the performance penalty of Round-Robin that exists in many situations.

C. In contrast, the Dynamic policy offers occasionally large performance benefits over RTC and exhibits a performance penalty only at very large overhead values.

We reiterate that in the comparisons presented so far we have used a potentially quite optimistic formulation of the static schedulers, which rely on detailed and accurate information about job runtime characteristics being supplied at job arrival time. In contrast, we have studied only the simplest version of Dynamic, one in which processors are released and requested immediately in response to changes in a job's parallelism. It is natural to wonder whether the performance of Dynamic can be improved even further by modifications that reduce the rate of processor reallocation. This question is the topic of the next section.

5. Improving the Dynamic Policy

The Dynamic policy defined in Section 4 makes processor requests and releases immediately in response to changes in job parallelism. Intuition leads one to believe that this causes a high rate of processor reallocations, and that consequently performance might be improved by taking a more conservative approach to reallocation.

With this in mind, we have examined two approaches to improving the performance of the Dynamic scheduler. The first, implemented in the system processor allocator, reduces the rate at which processors are acquired. It is a general mechanism that uses only simple measurements of a job's recent behavior, and is otherwise independent of specific knowledge about the nature of the job. The second approach, implemented in the application dependent task dispatcher, reduces the rate at which processors are released. Both approaches are discussed in more detail in the following subsections.

5.1. Reducing the Rate of Processor Acquisition

The purpose of this modification is to reduce the rate of "useless processor exchanges". A useless exchange is either the release of a processor shortly after acquisition or the acquisition of a processor shortly after release. Note that acquiring or releasing a number of processors in a short interval is not in itself an indication of a poorly behaved job. For instance, the MVA workload class has a tendency to exhibit this pattern even when it is running well.

We attempt to improve performance through modification of the processor allocator to reduce the rate of changes in the jobs' allocations. For each job j we keep a damping factor, d_j, whose initial value is 1. Whenever job j requests some number, p, of processors, the kernel attempts to allocate $\left\lceil \frac{p}{d_j} \right\rceil$ processors instead. We increase the value of d_j whenever a useless exchange is detected, and gradually decrease d_j otherwise. The particular manner in which this is done is as follows:

1. For each job j keep a timestamp of the last time the job *acquired* a processor.
2. On each processor *release* by job j, compare the current time to the timestamp for that job. If the difference is less than a context switch overhead time, d_j is set to the minimum of twice its current value and P, the total number of processors in the system. If the difference is more than a context switch time, d_j is set to the greater of half its current value and 1.

We double and halve d_j, rather than, say, adding and subtracting 1, in an attempt to allocate a number of processors that divides nearly evenly the amount of work left to be done. This avoids the situation, for instance, of an FJ(11) job having requested 10 processors (to bring it to a total of 11) and receiving only 9. Under the assumption that the tasks are of equal size, the 10 processors the job then holds cannot complete the work any faster than could the 6 processors obtained by allocating 10/2 additional processors, making this an expensive allocation for the system that is of no marginal benefit to the individual job.

5.2. Reducing the Rate of Processor Release

We rely on the task dispatcher to make sensible decisions about the appropriate time to release processors. Here we describe the modifications we have used for MVA, FJ, and VFJ jobs in an attempt to improve upon the basic "immediate release" policy.

5.2.1. The MVA Customized Task Dispatcher

Because of variation in individual task execution times, not all tasks in an individual level of the MVA computation complete at the same time. Because of synchronization constraints, there may be (short) periods of low parallelism even during the expanding parallelism phase of the computation. Releasing processors at these instants does not make sense (unless context switch overhead is extremely small), since they will be needed again by the job almost immediately. Thus, we modify the processor release policy so that processors are not released until the completion of the expanding parallelism phase of the MVA computation.

5.2.2. The FJ Customized Task Dispatcher

The cause of performance problems for the FJ workload is the policy for releasing processors during the sequential phase of computation. At moderate to high loads any processor released by a job j is likely to be quickly allocated to another job. When context switch overhead is large, more processing power is wasted in giving up processors at the onset of the sequential phase than in keeping them idle until they are needed in the next parallel phase.

We therefore modify the FJ task dispatcher so that it retains a parameterizable fraction of its maximal processor need even during sequential phases. For instance, if an FJ(11) job retains at the 50% level it does not release processors when it has 6 or less. If a job retains at the 100% level it never releases processors (except at job completion).

5.2.3. The VFJ Customized Task Dispatcher

The VFJ task dispatcher is modified in a manner nearly identical to that just described for the FJ jobs. As with the FJ task dispatcher, we define a parameter that restricts the number of processors the job will release. When a VFJ job forks it dynamically determines the target number of processors it would like to run with (see Section 2). The parameter of the dispatcher is a percentage that is multiplied (at each fork) by this target processor allocation. The result determines the number of processors the job will retain through the next sequential phase.

For instance, suppose the parameter is 50% and at the end of a sequential phase the job holds 3 processors and decides to ask for 5 more. Then it will hold 4 processors through the next sequential phase (assuming that it has acquired at least one more before then).

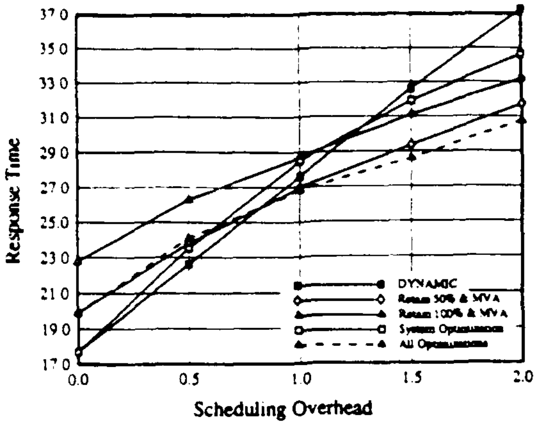


Figure 6 - Average Class Response Times with Modified Disciplines

5.3. Performance Results

Figures 6 and 7 show typical results obtained from these modifications to the two components of the scheduler. Figure 6 plots average job response time against context switch overhead for a multiple class workload (equal arrival rates of all six workload types) offering a total load of 50% useful processor utilization. (Results for other loads are nearly identical.) Figure 7 shows the response time of two individual classes in this multiclass environment. We plot results for the basic Dynamic policy (described in Section 3), as well as a number of the scheduler modifications just described. Under "System Optimization" the kernel processor allocator dampens changes in allocations (Section 5.1). However, jobs request and release processors immediately in response to changes in their parallelism, as in the base case. Under "Retain 50% & MVA" ("Retain 100% & MVA") the system processor allocator does not do any dampening, but the FJ and VFJ jobs guard 50% (100%) of their target number of processors. In these cases the MVA jobs follow the release policy described in Section 5.2.1. Finally, in "All Optimizations" we combine the System Optimization with the Retain 50% & MVA policy.

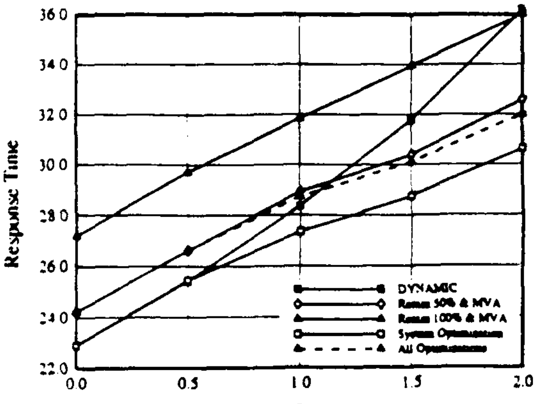


Figure 7a - MVA(10,10) Workload

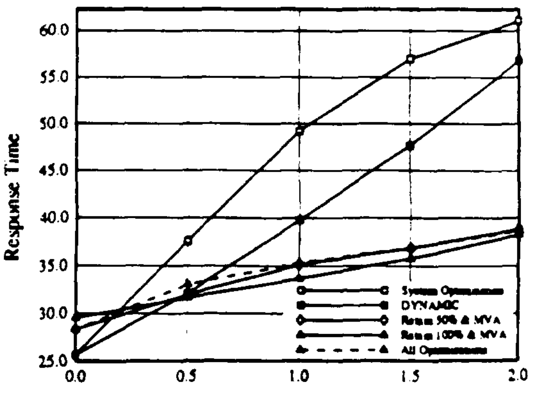


Figure 7b - FJ(11) Workload

Based on the results of Figures 6 and 7, as well as those of other experiments, we come to the following conclusions:

1. *The Basic Dynamic discipline is the best alternative over the low end of context switch overheads.*

   This is not unexpected, as when context switch costs are near 0 the basic algorithm makes most efficient use of the processors. We note that the range of context switch costs from 0.0 to no more than 1.0 are probably the most reasonable to expect in well designed applications.

2. *In general, the application specific modifications that reduce the rate at which processors are released are the best alternatives at higher context switch costs.*

   Looking at the overall average response times, the application specific modifications are preferable to the basic technique when overheads are greater than about 0.75. Examining each class individually, we see that the application specific modifications are preferable to the basic approach over roughly this same range.

   The overhead cost beyond which it is beneficial to retain a processor even if it is not immediately needed can be considerably shorter than the expected delay until the processor will be needed again by the job currently holding it. This can be seen in Figure 7b for the FJ(11) jobs. The mean delay until a released processor will be needed is at ~~least 1.0,~~ the task time of a sequential phase of service. However, retaining processors begins to help for context switch times beyond 0.5.

3. *Dampening the rate of allocations made by the system processor allocator is generally detrimental to performance.*

   Reducing the responsiveness of the system to processor allocation requests generally increases response time (Figure 6). The improvement found in the MVA(10,10) job class (Figure 7a) is bought at the price of a significant drop in performance for FJ(11) (Figure 7b): MVA(10,10) has more processors available to it because FJ(11) is prevented from acquiring them quickly.

   Combining the system level modification with the application specific modifications improves performance only very slightly over the application specific modifications alone. Given the performance risk involved if an FJ job, for instance, does not implement the application specific modifications, implementation of the system level modification seems ill advised.

The major conclusion of this section then is that for most applications (those with granularity large enough that context switch overhead is nearer 0.0 than 1.0) it does not make sense to increase the reaction time to processor allocation requests and releases. Thus, we conclude that as a system policy, the basic Dynamic scheme of Section 3 is the most suitable.

6. Conclusions

We have examined the performance of alternative two-level schedulers for medium scale, shared memory, UMA parallel machines. We have compared optimistic versions of static schedulers with a straightforward version of dynamic scheduling. From these comparisons we conclude that the Dynamic policy is the approach of choice unless overhead costs are very large. Of the two static policies, Round-Robin is not appropriate for general use on this class of machine since it improves upon the other static scheduler, Run-To-Completion, in only very restricted circumstances and often exhibits unstable behavior.

We have examined a number of natural approaches to improving the performance of the basic Dynamic scheduler. Each reduces the rate at which processor reallocations are made in an attempt to lower the context switch costs of this policy. We found that neither modifications to reduce the rate of processor acquisitions nor those to reduce the rate of processor releases were of general benefit.

Finally, we have compared two Fork-Join job structures, one (FJ) that partitions the parallel portion of the computation into a statically determined number of tasks and the other (VFJ) deciding on an appropriate partitioning at each fork. We found that the performance of VFJ was in general inferior to that of FJ. Improving the performance of VFJ seems to require a partitioning policy that estimates both the number of processors that will be available in the future and the future demand for those processors. Prospects for a reliable procedure of this sort are not bright, and so we conclude that FJ is probably the preferable structure in most circumstances.

References

[Almquist et al. 1989]
K. Almquist, R.J. Anderson, and E.D. Lazowska. The Measured Performance of Parallel Dynamic Programming Implementations. *Proc. 1989 Intl. Conf. on Par. Proc.* (Aug. 1989).

[Amdahl 1967]
G.M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *Proc. AFIPS 30* (1967).

[Bershad et al. 1988]
Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice & Experience 18*,8 (August 1988), pp. 713-732.

[Birrell 1989]
A.D. Birrell. An Introduction to Programming with Threads. DEC System Research Center (Jan. 1989).

[Coffman & Kleinrock 1968]
Edward G. Coffman, Jr., and Leonard Kleinrock. Computer Scheduling Methods and their Countermeasures. *Proc. 1968 Spring Joint Computer Conference*, pp. 11-21.

[Eager et al. 1989]
Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Trans. on Comp. C-38*,3 (March 1989).

[Graham 1966]
R.L. Graham. Bounds for Certain Multiprocessing Anomalies. *Bell Syst. Tech. J. 45* (1966).

[Kruskal & Weiss 1985]
Allocating Independent Subtasks on Parallel Processors, *IEEE Trans. on Soft. Eng. SE-11*, 10 (Oct. 1985).

[Lovett & Thakkar 1988]
T. Lovett and S. Thakkar. The Symmetry Multiprocessor System. *Proc. 1988 Intl. Conf. on Par. Proc.* (Aug. 1988).

[Majumdar & Eager 1988]
S. Majumdar and D.L. Eager. Scheduling in Multiprogrammed Parallel Systems. *Proc. 1988 ACM SIGMETRICS Conf. on Meas. and Mod. of Comp. Sys.* (May 1988).

[Ousterhout 1982]
J.K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proc. 3rd International Conference on Distributed Computing Systems* (October 1982), pp. 22-30.

[Polychronopoulos & Kuck 1987]
C.D. Polychronopoulos and D.J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. on Comp. C-36*,12 (December 1987), pp. 1425-1439.

[Reiser & Lavenberg 1980]
M. Reiser and S.S. Lavenberg. Mean Value Analysis of Closed Multichain Queueing Networks, *JACM 27*, 2 (April 1980).

[Sauer & Chandy 1979]
C.H. Sauer and K.M. Chandy. The Impact of Distributions and Disciplines on Multiple Processor Systems. *CACM 22*, 1 (Jan. 1979), pp. 25-34.

[Sevcik 1989]
K.C. Sevcik. Characterizations of Parallelism in Applications and Their Use in Scheduling. *Proc. 1989 ACM SIGMETRICS and Performance '89 Intl. Conf. on Meas. and Mod. of Computer Systems* (May 1989).

[Thacker et al. 1988]
C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite, Jr. Firefly: A Multiprocessor Workstation. *IEEE Trans. on Comp. 37*, 8 (Aug. 1988), pp. 909-920.

[Tucker & Gupta 1989]
A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. *Proc. 12th ACM Symp. on Op. Sys. Princ.* (Dec. 1989).

[Young et al. 1987]
M. Young et al. The Duality of memory and Communication in the Implementation of a Multiprocessor Operating System. *Proc. 11th ACM Symp. on Op. Sys. Prin.* (Nov. 1987).

[Zahorjan et al. 1988]
J. Zahorjan, E.D. Lazowska, and D.L. Eager. Spinning Versus Blocking in Parallel Systems with Uncertainty. *Proc. International Symposium on Performance of Distributed and Parallel Systems*, December 1988.

[Zahorjan et al. 1989]
J. Zahorjan, E.D. Lazowska, and D.L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. University of Washington, Tech. Rpt. 89-07-03 (July 1989).

Appendix A

Let there be a total of P processors to be allocated among J jobs. Let $T_j(p)$ as the execution time of job j with p processors, and define $T_j(0) = \sum_{i=1}^{J} T_i(1)$. We assume that $T_j(p)$ is convex in p, meaning that

$$T_j(p) - T_j(p+1) \le T_j(p-1) - T_j(p) \tag{A1}$$

for $p \ge 1$. This restriction is not unreasonable in practice, and allows a job to exhibit performance that at first improves with increasing processor allocation and then crosses a threshold beyond which performance degrades with increasing processor allocation.

Let $B_j(p)$ be the incremental benefit to job j of allocating it its p^{th} processor. Thus, $B_j(p) = T_j(p-1) - T_j(p)$ for $p \ge 1$.

An allocation A is a vector $(a_1, a_2, \cdots, a_J)$ such that $a_j \ge 0$, $1 \le j \le J$, and $\sum_{j=1}^{J} a_j = P$. (We do not allow allocations that assign no processors to a job when processors are available. In theory, one might delay a job in this way in anticipation of making a larger allocation in the near future than is currently possible.) Let E_A be J times the average response time that results from allocation A, that is,

$$E_A = \sum_{j=1}^{J} T_j(a_j) \tag{A2}$$

An allocation OPT is said to be optimal if $E_{OPT} \le E_A$ for every other allocation A.

Theorem:

Under the definitions given above, the greedy allocation followed by RTC is optimal.

Proof:

We observe that for any allocation A

$$E_A = \sum_{j=1}^{J} T_j(a_j) = -\sum_{j=1}^{J} \sum_{p=1}^{a_j} B_j(p) + \sum_{j=1}^{J} T_j(0) \tag{A3}$$

Thus, to minimize E_A we should maximize the double sum on the right hand side, since the other portion of the expression is independent of the allocation.

Imagine sorting the JP values $B_1(1), B_1(2), \cdots, B_1(P), B_2(1), \cdots, B_2(P), \cdots, B_J(P)$ from largest to smallest value. Call this list S. There must be such a sorted list for which $B_j(p)$ appears before $B_j(p+1)$ for $1 \le j \le J$ and $1 \le p \le P$ since, by (A1) and the definition of B, $B_j(p+1) \le B_j(p)$ for $p \ge 1$.

Now imagine following the greedy policy, that is, handing out P processors to the J jobs one processor at a time in a way that maximizes the benefit accrued by each processor allocated. This corresponds to giving the pth processor to the job j identified by the pth element of S. Greedy makes this choice because that benefit is at least as large as all the others. Further, because $B_j(n-1)$ precedes $B_j(n)$ in S, it must be possible to hand the pth processor to the job identified by the pth element of S. Thus, the greedy algorithm must make an optimal allocation.

Finally, we note that the greedy algorithm as defined will assign all P processors to jobs, and may therefore be required to assign so many processors to a job that its performance actually degrades with each added processor. A more reasonable scheme is for the system to retain processors for future arrivals if no current job can benefit from them. This is easily added to the greedy method and the proof by creating a dummy J+1st job such that $T_{J+1}(n)=0$ for $0 \le n \le P$. This job will be allocated processors by the greedy method in preference to any allocation that actually hurts some other job's performance.

Parallel Performance Analysis and Performance Tools

Quartz: A Tool for Tuning Parallel Program Performance

Thomas E. Anderson and Edward D. Lazowska

Department of Computer Science and Engineering
University of Washington
Seattle WA 98195

September 1989

Abstract

Initial implementations of parallel programs typically yield disappointing performance. Tuning to improve performance is thus a significant part of the parallel programming process. The effort required to tune a parallel program, and the level of performance that eventually is achieved, both depend heavily on the quality of the instrumentation that is available to the programmer.

This paper describes Quartz, a new tool for tuning parallel program performance on shared memory multiprocessors. The philosophy underlying Quartz was inspired by that of the sequential UNIX tool gprof: to appropriately direct the attention of the programmer by efficiently measuring just those factors that are most responsible for performance and by relating these metrics to one another and to the structure of the program. This philosophy is even more important in the parallel domain than in the sequential domain, because of the dramatically greater number of possible metrics and the dramatically increased complexity of program structures.

The principal metric of Quartz is the total processor time spent in each section of code along with the number of other processors that are concurrently busy when that section of code is being executed. Tied to the logical structure of the program, this correlation provides a "smoking gun" pointing towards those areas of the program most responsible for poor performance. This information can be acquired efficiently by checkpointing to memory the number of busy processors and the state of each processor, and then statistically sampling these using a dedicated processor.

In addition to describing the design rationale, functionality, and implementation of Quartz, the paper examines how Quartz would be used to solve a number of performance problems that have been reported as being frequently encountered, and describes a case study in which Quartz was used to significantly improve the performance of a CAD circuit verifier.

Index Terms – multiprocessor, performance, measurement, parallel programming, tuning

---

This material is based on work supported by the National Science Foundation (Grants No. CCR-8619663, CCR-8703049, and CCR-8700106), the Naval Ocean Systems Center, the Washington Technology Center, Digital Equipment Corporation (the Systems Research Center and the External Research Program), and IBM (a Graduate Fellowship).

Authors' address: Department of Computer Science and Engineering FR-35, University of Washington, Seattle WA 98195; (206) 545-2675; *tom/lazowska@cs.washington.edu*.

1. Introduction

The primary motivation behind building multiprocessors is to cost-effectively improve system performance. Even moderately increasing a uniprocessor's power can require substantial additional design effort as well as faster, and thus more expensive, hardware components. By contrast, once a scheme for interprocessor communication has been added to a uniprocessor design, the system's peak processing power can be increased linearly simply by adding processors. The incremental cost per processor has been reported to be as little as 15% of the initial system cost for small to moderate numbers of processors [Thacker et al. 1988], and larger but still close to linear for greater numbers of processors [BBN 1985; Pfister et al. 1985].

Of course, multiprocessors lose their advantage if this processing power is not effectively utilized, and while it is relatively easy to get good performance when there are multiple independent sequential job streams, it can be difficult to achieve good performance from parallel applications. The literature describes many attempts to parallelize algorithms and applications. (Burkhart and Millen [1989] survey some of this experience.) Typically, an initial implementation results in disappointing performance, but significant improvements can be obtained with subsequent effort. Sequent, for example, tells of a major customer whose first attempt at parallelizing a "dusty deck" resulted in a program that, given 8 processors, executed only 50% as fast as the original sequential program. After considerable effort by skilled engineers, nearly perfect speedup (a factor of nearly 8 on an 8 processor machine) was achieved [Rodgers 1986].

A major factor contributing to the large amount of skilled effort typically required to achieve good parallel program performance is the shortage of good performance analysis tools. In the absence of such tools, performance problems must be identified through a combination of guesswork, folklore, and application-specific instrumentation. The object of this paper is the design rationale, functionality, implementation, and use of a new tool for tuning parallel program performance.

The philosophy underlying our work is that an effective tool for tuning parallel program performance must be based on a clear view of the causes of poor performance, and on a specific methodology for improving that performance. By being selective about what it measures and presents, the tool can focus the programmer's attention on the information needed to tune performance, eliding details about second-order effects. Measurement efficiency also is improved by designing the tool to record just the important behavior.

Selectivity is possible because, although parallel performance in general is much more complex than sequential performance, experience (discussed in Section 3.2) suggests that poor parallel performance typically arises from a relatively small number of factors. For applications whose performance is dominated by periods of limited parallelism, the tool should identify those sections of code that account for most of this time so that these sections can either be re-structured to increase concurrency or optimized to reduce their impact on overall performance. Time spent spin- (or "busy"-) waiting must be correctly represented, since spinning processors appear to be busy even though they are not computing useful results. Finally, for applications with large amounts of real parallelism, performance can only be improved by optimizing (but not further parallelizing) the code that executes for the greatest proportion of time.

Based on these observations, we propose a new way to view parallel program performance on shared memory multiprocessors. We focus on the total processor time devoted to each section of code, subdivided according to the distribution of the number of other processors concurrently busy (as opposed to idle or spin-waiting) when that section of code is being executed. Routines can be compared by considering their processor time divided by the concurrent parallelism. This usually reflects their relative importance to program performance: a routine that executes while no other processors are busy can be responsible for a large percentage of the runtime of a program, even though it uses only a small fraction of the total processor time. Furthermore, this measure indicates whether performance can be improved by re-structuring to increase parallelism, or only by simple optimization.

We tie this measure to the logical structure of the program's procedures. Good engineering practice demands that large programs, whether sequential or parallel, be structured using hierarchical abstractions [Graham et al. 1982]. We report our performance measures for each procedure and for all the work done on its behalf, either synchronously via a normal procedure call or asynchronously through parallelization. The programmer can use this to walk through the hierarchy, focusing on just those procedures that, along with their children, account for most of the poor performance. We expect that for parallel programs as for sequential ones, a relatively small proportion of the code will be responsible for most of the runtime.

These measurements can be made efficiently on a shared memory multiprocessor by checkpointing to memory the number of busy processors and the state of each processor, and then statistically sampling this information using a dedicated processor.

We have developed a tool to test these ideas, called *Quartz*. Quartz was built by modifying an application-level thread package similar to the one described in [Anderson et al. 1989]. Quartz uses only the normal profiling support available on UNIX-like shared memory multiprocessors; it currently runs on the Sequent Symmetry multiprocessor [Sequent 1988].

The remainder of the paper discusses these ideas in more detail. Section 2 examines existing measurement tools for tuning program performance. Section 3 describes Quartz: its motivation, its functionality, its applicability to a number of performance problems that have been reported as being frequently encountered, and its implementation. Section 4 describes a case study in the use of Quartz to improve the performance of a specific parallel application, a CAD circuit verifier. Section 5 considers the implications of our work for the monitoring of sequential programs and non-shared-memory multiprocessors. Section 6 summarizes our results.

2. Existing Tools for Tuning Program Performance

2.1. Tools for Sequential Programs

The philosophy underlying Quartz owes much to the experience of UNIX gprof [Graham et al. 1982], a tool for tuning the performance of sequential programs running on uniprocessors.

Years of experience tuning sequential programs indicate that the major difficulty is focus: it is relatively easy for the programmer to improve the processing time of a small section of code, but lots of effort is commonly wasted in the wrong places – tweaking code that has only a small impact on overall performance.

Gprof's solution is to highlight the "hot spots" of the program, and to do so in a way that exploits the hierarchical structure of large programs. Gprof presents to the programmer the total processor time of each procedure, including time spent on its behalf if it calls other routines. With this information, the programmer can tune the program in a top-down fashion, focusing effort on those functions that have the greatest impact on performance.

Gprof is relatively efficient. It periodically interrupts the program to sample the program counter, thereby estimating the execution time of each procedure. While sampling produces only an estimate, the approach is most accurate just where it needs to be: for those routines where the program spends most of its time. Gprof also collects the call graph: who called whom how many times. This is done by using compiler support that makes each procedure execute a monitoring routine during its prologue. Gprof then computes its central metric, the processor time spent on a procedure's behalf, by making the assumption that all calls to the same procedure take the same amount of time. Processing time is propagated bottom-up from callee to caller according to the caller's proportion of the total calls.

Gprof seems so natural in retrospect that it is easy to forget the alternative approach taken by a number of other tools: to (expensively) measure everything that could conceivably be of relevance to program performance, and to report these measurements without concern for how they relate to each other or to the structure of the program.

Our goal for Quartz was to achieve a tool for tuning parallel program performance that is analogous to gprof in that it efficiently measures exactly what is important, and relates these measurements to one another and to the structure of the program. This philosophy is even more important in the parallel domain than in the sequential domain, because of the dramatically greater number of performance metrics and the dramatically increased complexity of program structures. The next two sub-sections discuss, in this context, existing approaches to tools for tuning parallel program performance.

2.2. Non-Integrated Tools for Parallel Programs

Many useful measures of parallel program performance have been proposed. Each provides a view of some important aspect of program behavior. However, in many existing tools, their lack of integration with each other and with program structure limits their usefulness.

Perhaps the simplest approach to parallel program measurement is to extend sequential UNIX gprof to run on a multiprocessor. In place of processor time on one processor, multiprocessor gprof measures the sum of the time spent on each processor [Aral & Gertner 1988]. Unfortunately, as we have noted, a procedure's total processor time is not related in a simple way to parallel runtime. Something more than a straightforward adaptation of sequential UNIX gprof clearly is necessary.

Another common tool displays the number of busy processors across time by periodically sampling the number of runnable processes. Assuming that all activity is due to the program in question, this allows the programmer to see if there are periods of time when there is too little parallelism to keep all the processors busy [Halstead 1986]. A significant shortcoming of tools like this is that it can be difficult to relate the periods of poor parallelism to specific sections of code that can be changed. Further, the fact that some processors are not doing useful work can be concealed, since spin-waiting processors appear to be busy.

The DEC SRC Firefly has a tool that measures the time spent waiting for each lock protecting a shared data structure [Thacker et al. 1988]. A lock ensures that threads manipulating the shared data structure have mutually exclusive access to it. This serial execution can limit performance. By measuring the wait time, the tool can determine which critical sections are the worst bottlenecks; these can then be restructured to increase parallelism. This information is useful, but long waits for a lock will not affect performance if there is always other work to do during the wait, and monitoring a lock can increase the length of time that the lock is held, creating "artificial bottlenecks" when monitoring is enabled.

Quartz provides many of the same metrics as these tools, but correlates the metrics to one another and to the structure of the program. For example, Quartz measures not only how many processors are busy, but also which procedures execute during periods of low and high parallelism.

2.3. Trace-Based Tools for Parallel Programs

The issue of determining in advance exactly what information will be needed to tune the performance of a parallel program can be finessed by recording a trace of every interprocessor synchronization event with a timestamp of when the event occurred. The behavior of the program can be completely reconstructed from such a trace [Fowler et al. 1988]. Arbitrary metrics (whether general or application-specific) can be computed using a common interface to the trace data. Finally, the metrics obtained from the trace can be integrated with each other and with the structure of the program.

One drawback to this approach is that both the collection and the post-processing of trace files is expensive. For programs that perform frequent synchronization, the trace files can be prohibitively large. Consider a program running on a hypothetical shared memory multiprocessor with 20 5-MIPS processors, each of which on average performs a monitorable event (such as acquiring or releasing a spin lock) every 500 instructions, and where each event record is 10 bytes. If the program being monitored executes for one minute, the trace file will be 100 megabytes. Similar estimates appear in [Malony et al. 1989] to justify hardware support for recording traces.

Nevertheless, tools have been developed that collect trace data and post-process it into useful measures. We argue later that the much of the information provided by these tools can be measured or approximated more efficiently by sampling.

Monit [Kerola & Schwetman 1987], for example, uses a trace file to compute the behavior of higher-level objects, such as the number of busy processors or the number of threads waiting to enter each critical section. The behavior of each object is then plotted on a timeline. After identifying those phases of execution with few busy processors, the programmer can visually correlate these phases to the behavior of other objects (discovering, for example, that parallelism is low while a specific critical section has a large number of waiting threads).

Although Monit's display is at a higher level than the raw trace data, it still can present a massive amount of data to the programmer. Only a few timelines will fit on a screen at a time, but Monit provides little help in identifying those containing information relevant to the measured lack of parallelism. As the complexity of the application increases, so does the number of objects to monitor, making focusing more important.

IPS [Miller & Yang 1987] attempts both to guide the programmer to performance problems and to provide useful statistics about those problems. Its central focusing metric is the time each process and procedure spends executing the critical path. The critical path is the longest path through the task graph – the series of sequential pieces of code (that cannot be done in parallel because they communicate one to the next) that takes the longest to execute. The elapsed execution time of the program can be reduced only by shortening the length of the critical path.

One drawback to critical path analysis is its expense: it requires a complete trace of all interprocessor communication. (To be fair, IPS was originally designed to measure programs running on local area networks of uniprocessors. Because of the high latency and low bandwidth communication on these systems, only programs with relatively infrequent synchronization can run efficiently. Under these conditions, the size of the trace file would be manageable.) Yet critical path analysis is still just a heuristic: there is no guarantee that reducing the critical path will actually reduce the execution time of the program. There may be another path through the task graph with almost the same length that will be unaffected by the change. Critical path analysis also does not indicate *how* to reduce a procedure's completion time. One way is to reduce its sequential execution time. Another is to parallelize it. But parallelization will only be of benefit if there are idle processors to exploit when the procedure runs.

3. Quartz: Its Functionality, Applicability, and Implementation

Our goals for a tool for tuning parallel program performance are:

- It should identify the sections of source code most responsible for poor performance.
- It should present its measurements hierarchically, to allow top-down tuning according to the logical structure of the program.
- It should measure parallelism (properly representing spin-waiting as a loss of parallelism) and it should tie this to the source code, identifying where re-structuring to increase parallelism is necessary and where code optimization is appropriate.
- It should measure program behavior in sufficient detail to provide some insight into the type of re-structuring that will work.
- It should do all this efficiently and without significantly affecting the behavior of the measured program.

Quartz meets these criteria. Before describing it, we must define some terms. A thread (or "lightweight process") is a sequential execution stream; it is the basic unit of parallel work. A thread starts another thread by giving it a procedure to run; the initial thread continues in parallel with the created thread. Thread creation is thus essentially an asynchronous procedure call. If threads are implemented as part of an application library, they can be within an order of magnitude of the cost of a procedure call [Anderson et al. 1989]; they can thus be used for procedure-level parallelism.

Threads can synchronize with one another. One type of synchronization object is a lock, used to ensure mutually exclusive access to a shared data structure. Another is a condition or barrier used to enforce a data dependency, as when one thread reads data produced by some other thread. In both cases, synchronization may cause the thread to wait, either because the lock is busy or because the data it requires has not been produced. Since there may be more threads than processors, a waiting thread has a choice: either spin until the lock is free or the data is available, or block, relinquishing the processor to run another thread. Thus, there is a difference between a program's *effective* parallelism, the number of busy (not idle or spinning) processors, and its *nominal* parallelism, the number of runnable threads, some of which may be spinning. Our measurements refer to the activity of just the processors executing the application, and not any processors concurrently executing other applications.

The remainder of this section is divided into three sub-sections. The first describes the functionality of Quartz: the specific metrics that it reports. The second shows how these metrics can be used to detect and fix a number of performance problems that have been identified by others as commonly occurring. The third provides an overview of the implementation of Quartz. A case study in which Quartz was used to tune a specific application is described in Section 4.

3.1. The Functionality of Quartz

The principal measurement made by Quartz is each procedure's total processor time, subdivided according to the distribution of the effective parallelism while it is executing. This measurement provides the basis for comparing procedures according to their effect on overall performance. Quartz computes a weighting factor for each procedure by dividing each level of effective parallelism into the corresponding processor time value, and summing these ratios. To understand the rationale for this weighting factor, consider a program with two functions, one that always executes sequentially when no other processors are busy and one that computes its result completely in parallel. If each function takes the same total processor time, the sequential one requires a factor of P greater elapsed time (where P is the number of processors) and will have a much greater impact on program performance. If the two functions take the same elapsed time, then the same percentage improvements in either will have equal impact on performance. Further, knowing the concurrent effective parallelism while a routine executes is more important than knowing the effective parallelism it generates: a serial routine that is always overlapped with other computation will have little affect on performance compared to a serial routine that always executes by itself. Our weighting factor reflects these observations.

For each procedure, synchronization object, and thread, and for the work done on its behalf:

Processor time, split according to the distribution of the number of other processors concurrently busy during the execution.

A weighted sum equal to the processor time divided by the number of concurrently busy processors.

The elapsed time spent in each state (busy, spinning, blocked, or ready), along with the distribution of the number of runnable threads while it is in that state.

Table 1: Principal Performance Measurements in Quartz

To focus the programmer's attention on those areas of the program that have the greatest impact on performance, we present a list of procedures sorted according to their weight plus the weight of the work done on their behalf. This includes work done synchronously or asynchronously (via threads). The program's main procedure, then, has a weight equal to the elapsed time of the program; the functions it calls to do the work of the program divide that weight among them. Quartz's ordering is analogous to what gprof does with processor time, except that Quartz uses a weighting function related to parallel performance. In both Quartz and gprof, the programmer can trace performance top-down through the program.

The weighting function indicates *where* improvements can be made. Quartz also provides information about *what* can be done to improve performance. Part of this information comes from the measurement of processor time by parallelism. This metric indicates whether performance can be improved by re-structuring to increase effective parallelism, or only by simple optimization. Certainly, procedures that always execute when all processors are busy will not benefit from further parallelism.

Given that re-structuring is necessary, an accounting of the *elapsed* time spent by a procedure can help identify what kind of re-structuring is most likely to succeed. Quartz measures each procedure's elapsed time spent busy, spinning, blocked, or ready to run, along with the distribution of the number of threads that are available to run while the procedure is in each state. For example, if a procedure is busy executing and there are few other busy processors, then the nominal parallelism indicates whether the other processors are idle or spinning. If idle, then performance can be improved by parallelizing the procedure (creating threads to do its work), provided this is possible. If spinning, then there is no benefit to creating more threads. Similarly, threads that are blocked or spinning represent deferred work; if the program were re-structured to reduce or eliminate waiting, then parallelism would increase.

Quartz makes the above measurements separately for each procedure, synchronization object, and thread. Measurement based purely on procedures would ignore the fact that parallel performance can depend more on the data object passed to a procedure than on the implementation of the procedure itself. It is only mildly interesting to know the total time spent waiting at all barriers; it is much more useful to know that some specific barrier accounted for most of that time. As a special case, the time spent executing in a critical section is attributed to the lock on that critical section. (Quartz also measures queue length distributions for synchronization objects.) Per-thread information allows us to determine if different threads executing the same procedure (on different data objects) have different performance.

By measuring synchronization objects and threads in the same way as procedures, we can present the programmer a uniform focusing metric. All are sorted in the same list by their measured weight to simplify tracing performance through the program. For example, if contention for a lock determines performance, the lock object will have a high weight since its critical section is always executing while few other processors are. (Spin time is factored out in computing weight.)

An important difference from gprof is that we explicitly measure the work done on behalf of a procedure or lock object. Gprof explicitly measures only the work done within each procedure, making the assumption that the processor time of its children is independent of who called them. Gprof uses the call graph (who called whom, and how many times) to propagate processor times from callee to caller according to the caller's percentage of the total calls to the callee. We cannot make a similar assumption. The effective and nominal parallelism while a procedure executes depend not only on what that procedure does, but also on the parallelism when it is called. Different calls to a low-level formatting routine might have vastly different concurrent parallelism. Still, even though it is not useful for propagating our measurements, Quartz does record the call graph (including calls to/from synchronization objects) to help the programmer in tracing performance top-down through the program.

3.2. Detecting Frequently Encountered Performance Problems Using Quartz

In this section, we argue by example that Quartz is useful for detecting and fixing a number of common parallel program performance problems. We asserted in Section 1 that although parallel performance in general is much more complex than sequential performance, experience suggests that poor parallel performance typically arises from a relatively small number of factors. One piece of evidence for this is Table 2, which lists the performance problems most frequently encountered by three "vendors" of parallel computing systems who participated in a working group concerned with "Sources of

Performance Degradation" at the *NSF/CMU Workshop on Performance-Efficient Parallel Programming* in 1986. The key observation is that none of the problems involve subtle timing issues that might require a complete trace of synchronization activity.

Sequent

1. A problem decomposition that puts most of the work in one thread (e.g., the optimizing phase of a concurrent compiler or a "busy" region in a ray-tracing algorithm), so that little real concurrency can be realized.
2. Memory thrashing due to a poor choice of operating system parameters.
3. Excessive I/O that is not overlapped with computation.
4. A synchronous software structure, such as might arise from a very large granularity, a producer-consumer relationship with a small number of buffers, or the use of an unnecessarily restrictive synchronization construct (e.g., barriers where critical sections would suffice).

Harris

1. Synchronization overhead.
2. Contention for shared variables, including counting semaphores, task queues, and the "problem heap".
3. Starvation due to a small problem size.

CMU Warp

1. Excessive I/O that is not overlapped with computation.
2. Data dependencies in loops.

Table 2: Frequently Encountered Performance Problems
(NSF/CMU Workshop on Performance-Efficient Parallel Programming)

The first issue in tuning any parallel program is to identify which segments are responsible for the poor performance. As with sequential programs, we would expect that of the large number of functions in a parallel program, a relative few will be responsible for most of the program's runtime. By computing a weight based on both processor time and parallelism, and by accounting for all of the activity done on behalf of a function, Quartz allows the programmer to walk through the program hierarchy to find those few functions. Once the general area of difficulty has been located, the approach to tuning depends on the situation:

3.2.1. Functional Decomposition

Some computations have several functionally distinct parts, each assigned to a distinct processor. An example of this is a pipelined compiler: separate threads (and processors) execute the scanner, parser, optimizer, and code generator, streaming results one to the next (Sequent #1 in Table 2). Performance difficulties usually relate to load imbalance. If one phase has more work to do than the others, the others must sit idle waiting for it. If the optimizer is the bottleneck, the scanner and parser will have to wait for buffer space to forward their partial results, while the code generator must also periodically wait for results to be completed.

Quartz would identify this problem: the thread executing the optimizer would have a longer execution time, spend more time executing when few other processors are busy, and thus have a larger weight, than the threads executing the other phases. Other tools would also handle this case. For example, a display of processor activity across time would show that the processor executing the optimizer was always busy, while the other processors sometimes wait. (Of course, many tools that display processor activity fail to relate processors to procedures.) Similarly, a critical path analysis would show that the execution of the optimizer constituted most of the critical path.

It is also easy with Quartz to identify the phase that is the secondary bottleneck – in the compiler example, the one that would limit performance if the optimizer were improved. If the code generator ran for almost as long as the optimizer, then it would be given slightly less weight, indicating that attention should be focused on improving both phases. It is difficult to extract this information from a timeline, since all phases but the optimizer periodically block. Critical path analysis only identifies the primary bottleneck, so iteration would be required.

Another performance problem with pipelines is starvation (Harris #3). This occurs if the problem size is small relative to the time for each phase to start streaming results. In this case, the later phases spend much of the total time waiting to start; the earlier phases finish well before the program completes. Quartz would show that each phase spends much of its *elapsed* time idle. (The weighting function would highlight the first and last stages, since their work is the least overlapped with other stages.) A solution is to reduce the time before each phase starts streaming its first results.

3.2.2. Data Decomposition

Some programs compute the same function on many pieces of data. These programs can be parallelized by assigning different pieces of data to different processors. Unlike functional decomposition, each processor executes the same function at the same time. Again, a frequent issue is load balancing: the required computation may vary widely for different pieces of data. An example of this is ray-tracing where part of the picture has the majority of the activity (Sequent #1); another is a fluid dynamics computation where turbulence is concentrated in certain regions. In such situations, performance is limited by the processor assigned to the data regions with the longest execution times.

Whether a different thread is used to run the function on each object, or on collections of objects, Quartz will show if the execution times are balanced. If they are not, one of the threads will execute while other processors are idle, and there will be long average queue lengths at the barrier that checks that all threads have completed before continuing.

It can be difficult to relate the performance of a thread to the symbolic names of the data objects it works on, particularly in conventional (non-object-oriented) languages. For instance, the procedure a thread is to execute can be passed an index that only implicitly refers to the object it is to work on. As a result, we rely on the programmer to make this connection by providing a symbolic name when each thread is created. In an object-oriented language such as C++ we could extend Quartz not only to keep track of the symbolic names of data objects passed to threads, but also to explicitly take measurements for each procedure-data object pair, to allow both an object-oriented and a procedure-oriented view of performance. We intend to port Quartz to Presto [Bershad et al. 1988], a C++ based parallel programming system, to further explore this topic.

3.2.3. Synchronization

The need to synchronize the work of different processors can cause another class of performance problems. For instance, execution time is increased by the overhead of parallelizing the job: distributing work to various processors, serializing access to shared data structures, and enforcing data dependencies (Harris #1). Even if the program is perfectly parallel, this time can dominate. Fortunately, it is easy to measure. If there is a sequential version of the program, its functions will likely correspond to functions in the parallel version, and the execution time of each function can be directly compared to determine the effect of overhead. (The execution time added by monitoring must of course be factored out.) Alternately, given measurements of the performance of the thread package, the number of calls to each thread function, such as to create a thread or to acquire a lock, can be used to compute overhead.

Performance can also be affected by waiting for data dependencies to be satisfied (Warp #2; Sequent #4) or for access to a busy critical section (Harris #2). Waiting threads represent deferred parallelism; Quartz identifies this by measuring queue lengths and the average wait time (the total elapsed time spent waiting divided by the number of accesses to that synchronization object). For example, if a loop data dependency limits parallelism, there will be a long queue length at the point where the data dependency is enforced. Note that two of the examples of contention cited in Table 2 are for locks within the thread package; we measure contention for these locks in the same way that we measure locks in the application code.

Even if there are many threads waiting on a synchronization object, the question of whether it makes sense to re-structure the program to release that parallelism depends on whether the time is spent spinning or blocked, and on the nominal parallelism. When there are at least as many runnable threads as processors, blocked threads have no impact on performance beyond the initial context switch. Re-structuring to increase the number of ready threads does not help in this case. By contrast, spin-waiting always wastes processing cycles, regardless of the number of runnable threads, but if there are excess runnable threads then performance could be improved by blocking instead of spinning.

If re-structuring is necessary, the number of threads waiting at a lock can be decreased by any of: reducing the number of accesses (from the call graph), thereby reducing contention; decreasing the size of the critical section (its busy time); distributing accesses more evenly across time (if the queue length is sometimes zero and sometimes very long); or modifying the protected data structure to allow parallel accesses (for example, by giving each processor a separate copy).

Waiting for data dependencies can be reduced by computing the data earlier, or, if an overly restrictive synchronization construct was used, by allowing the thread to continue temporarily without it. Fuzzy barriers are a special case of the latter [Gupta 1989].

3.2.4. Input/Output

The time spent doing I/O was mentioned twice in Table 2 (Sequent #2, Warp #1). If a program reads a significant amount of data from an I/O device, then the reads should be overlapped with the computation; in other words, the reads should be started early so that they complete before the data is needed. The natural style, however, is synchronous: when the data is referenced, start a disk read and wait until it returns.

As a result of the operating system interface on the Sequent, the current implementation of Quartz measures time spent doing I/O as processor time, attributed to the procedure that performs the I/O. If the I/O is not overlapped, the time spent waiting in the kernel will be increased in weight because of the processors waiting for the I/O to finish. Given kernel support for threads, Quartz could monitor the kernel disk queue as a normal synchronization object.

If a program spends a lot of time doing disk accesses, it may benefit from exploiting parallelism in the disk sub-system. Tuning a program's use of parallel disks is in many ways similar to tuning its use of parallel processors, although initial file placement is an issue as well. We expect that some of the techniques we have described in this paper could be applied to this problem.

3.2.5. Limitations

We have designed Quartz to measure only those aspects of program behavior that are needed to detect and fix frequently occurring parallel performance problems. The tradeoff is that Quartz therefore does not help with every performance problem that can occur in parallel applications.

When threads execute at the same time, Quartz weights each equally even though only one is on the critical path. As an example, consider a program with a critical section that restricts parallelism. The processor time spent executing outside of the critical section can appear important, even though reducing or parallelizing it will have no effect on program's performance. Although this can seem anomalous, Quartz's metric can help in this case by identifying code that may be a secondary bottleneck. Similarly, Quartz does not measure thread scheduling decisions (although problems can sometimes be identified, for instance, if a thread spends a long time waiting for a processor and then executes serially) or contention for the bus or memory, even though these can affect performance.

3.3. The Implementation of Quartz

We have implemented Quartz on a Sequent Symmetry shared memory multiprocessor [Sequent 1988]. The Sequent runs DYNIX, a multiprocessor adaptation of UNIX. Since DYNIX processes are too expensive to use directly as threads, we built our system by adding monitoring code to a thread package similar to the one described in [Anderson et al. 1989]. That thread package works by creating a DYNIX process for each processor, and then multiplexing threads onto the DYNIX processes. Our implementation did not modify DYNIX or the C compiler; it used only the support they provide for gprof.

Our implementation addresses the twin concerns of efficiency and accuracy. Because program tuning is iterative and interactive, a tool's usability depends on the elapsed time from program compilation to report production. Accuracy is trickier. Unlike sequential programs where the execution overhead due to monitoring is easily factored out, a change to a parallel program can alter its behavior in subtle ways. For instance, monitoring code that increases the time that a lock is held may increase the contention for the lock. Analogously, instrumentation added outside of a critical section will cause a net decrease in the contention for that critical section.

Our approach is to use statistical sampling by a dedicated processor. A set of processors executes the program normally, maintaining their state in shared memory by special code executed during thread operations and at procedure entry and exit. This state is then sampled by a dedicated processor that does not participate in executing the program. We impose no synchronization beyond hardware interlocks between the sampling processor and the other processors; rarely-accessed locks are used by the normally executing processors in building the call graph. (We sample by means of a dedicated processor rather than interrupts because interrupts cannot provide accurate correlations between processor state and overall program state. On the Sequent, as with most multiprocessors, interrupts are fielded by each processor asynchronously; by the time the program state is sampled, it may have changed in a way affected by the fact that there was an interrupt. For example, if the interrupted processor is holding a lock, the queue length at the lock will be greater than a purely random sample would indicate. Similarly, measuring a procedure's execution time directly with timestamps does not allow us to correlate that time to the number of busy processors.)

The nominal and effective parallelism are maintained in centralized counters, updated when a thread is added to the ready queue and when a processor becomes idle or starts or stops spinning. The counters are maintained with atomic increment and decrement instructions, to avoid making access to them a bottleneck. Most multiprocessors, including the Sequent, support such instructions.

In addition to the execution stack, we maintain a profile stack of monitored procedures for each thread. This allows us to record both the time spent in a procedure and the time spent on behalf of the procedure. The dedicated processor copies the number of busy and ready threads, copies the profile stack, and then bumps the appropriate measurement record for each different procedure on the stack. (Recursive procedures are counted only once.) While the stack may have changed between recording the number of busy threads and copying the stack, reducing consistency, sampling itself is only an approximation. We do not lock the profile stack to prevent changes from occurring; this would unnecessarily perturb the execution of the program. Note that locking would be harder to avoid if we were to sample from the execution stack since that would require tracing the chain of frame pointers.

The profile stacks are of fixed size, established at compile time. Overflows are caught, prevented, and later reported to the programmer. We expect that overflows will occur only rarely, since we push a procedure onto the stack only if the previous entry is different, eliminating immediately recursive calls, the most common cause of arbitrarily deep stacks. This also has the effect of reducing the work of the sampling processor.

We use the normal compiler support provided for gprof. A monitoring routine is called in the prologue of each profiled procedure. Exactly as gprof does, we use this routine to update the count of calls to the procedure from its caller; we also push the procedure onto the profile stack. Because the compiler inserts only a prologue call, we manipulate the execution stack so that when the procedure returns, it returns first to our code that pops the profile stack, and then to the caller procedure. This is a bit inefficient, but easy to implement.

To simplify mapping from the entries on the profile stack to the measurement data for each procedure, we assign each procedure a unique ID. The gprof monitoring routine is passed a pointer to a procedure-specific location; this was originally used to count the number of calls to the procedure. After the program has been linked, we modify the object file so that each procedure's location holds a unique ID; this ID is what is pushed onto the stack and used to index the procedure's data record.

The synchronization routines in our thread library are specially modified. Each object has a data record containing the call graph and execution time information. When the object is accessed, the call graph is updated and a pointer to the synchronization object is pushed; the pointer is popped when the thread no longer must wait. Locks are handled as a special case. Normally, the procedure that acquires a lock is the one that releases it, in which case we are safe to push the lock before it is accessed and pop it after it is released. This attributes the time spent waiting for and holding the lock to the lock object, and only adds two instructions to the inside of the critical section: setting the state of the thread to no longer spinning, and incrementing the number of busy processors.

When a thread is created, we copy the profile stack from the creating thread to the new thread. This allows the sampling processor to attribute execution time across asynchronous procedure calls.

Our system does not currently provide for interactive control of which routines are to be profiled. This would be easy to add, but in truth, we doubt that it is the right approach. Aral and Gertner [1988] argue that gprof's overhead is too high to allow only compile-time control. They use this to motivate Parasight, a system for execution-time code modification and re-linking. But the overhead of gprof, and of our system, could be dramatically reduced with a small amount of compiler support. For example, most of the time in gprof is spent building the call graph; it crawls up the execution stack to find the caller address, hashes on it, checks the callee address, etc. A simpler method is to determine caller-callee pairs at compile time and to simply bump a statically allocated counter before each call. Calls made via function pointers, a rarer case, could use the current, slower approach.

4. A Case Study: Using Quartz to Tune a CAD Circuit Verifier

We argued "abstractly" in Section 3.2 that Quartz is well-suited to detecting and fixing a spectrum of parallel program performance problems that have been identified by others as commonly occurring. Of course, the crucial question is whether Quartz is an effective tool in practice. In this section, we describe our experience in using Quartz to tune an existing parallel application.

The application we tuned, called Verify [Ma et al. 1987], compares two different circuit implementations to determine whether they are functionally (Boolean) equivalent. It was written for a dissertation to demonstrate that an existing production CAD program could be parallelized with good speedup. The program has 2900 lines of C code, and was written for a Sequent Balance with twelve processors. The circuits we used as inputs in our tests were combinational benchmarks for evaluating test generation algorithms.

The initial speedup of Verify on our Sequent Symmetry was already good: 9.0 using 18 processors. (Because no sequential version of the program was available, speedup was measured as the time to run the program (including process creation and I/O) on one processor divided by the time to run it on 18 processors.) Even though neither of us was familiar with the program or with CAD algorithms in general, over the course of several hours we were able to improve its performance by 40%. Its initial runtime on one processor was 112 seconds; with our changes, the runtime dropped to 7.7 seconds on 18 processors.

Most of this improvement came within the first few minutes of using Quartz, demonstrating the utility of our weighting function. For program verification purposes, Verify logged a trace of shared data structures as they were created. This accounted for less than 2% of total processor time according to gprof – an acceptable overhead. However, Quartz showed that this logging actually accounted for 25% of the program's runtime, because it occurred during the sequential initialization phase of the program. The importance of reflecting the program structure by reporting the work done on behalf of a procedure was demonstrated here, since the time was split among several lower-level procedures called by the logging function. When logging was removed, the program's speedup improved from 9.0 to 12.2.

When we re-profiled the modified program, Quartz showed that 10% of the program's runtime was still spent in the sequential initialization phase. To reduce this, we read in and allocated data structures for the two input circuits in parallel with each other and with the operating system process creation needed to start the program running on all 18 processors. This improved performance somewhat to a speedup of 12.7.

At this point, we stopped trying to further parallelize the program. Once a program's speedup is high, further improvements become much more difficult. The routines responsible for the difference from ideal speedup account for only a small fraction of the total program runtime; thus even radical improvements to these routines can reduce overall runtime only slightly.

In our case, Quartz showed that over 80% of the program's runtime was now being spent executing entirely in parallel. The remaining time was split between processor starvation during initialization and termination. During initialization, Quartz showed that performance was limited by the fact that the input files were not balanced (one circuit was a minimization of the other). During termination, the problem was that some processors finished early and had to wait for the rest to finish; dividing the problem into smaller size sub-problems might help this problem. Fixing these problems seemed to be more effort than it was worth.

Instead, we noted that small changes in the routines that account for most of the parallel execution time would have a large relative effect on runtime. Two routines accounted for over half the execution time of the program, and we were able to improve program performance by another 12% with a few quick tweaks to these routines. (These routines were already highly tuned from the original program.)

A major motivation behind Quartz is efficiency. We measured the overhead Quartz added to this application. Quartz increased the runtime of Verify by roughly the same amount as gprof: about 70%. Even though Quartz does more work than gprof on each procedure call and synchronization event, this is made up for by off-loading sampling to a dedicated processor. Verify makes stringent demands on Quartz: it generates roughly 9 million procedural and synchronization events (1 million per second when running on 18 processors). Even with 18 processors running, the dedicated sampling processor was able to sample each processor's activity every 3 milliseconds.

Something that we did not expect was demonstrated by using Quartz on a real application: there is less "performance locality" in parallel programs than in sequential ones. The top eight procedures account for 95% of the elapsed time of Verify on one processor. With 18 processors, though, it takes over 20 procedures to reach the same 95% level. In retrospect, the reason is obvious. The routines that account for most of the time on one processor are parallelized and therefore account for much less of the program's runtime on multiple processors; at the same time, routines that take only a small amount of processor time can become important if they run sequentially.

5. Implications for Other Systems

While we have implemented Quartz on a shared memory multiprocessor, our work has implications for other systems.

On multiprocessors with distributed memory, such as the Intel Hypercube, a dedicated sampling processor would not have efficient access to the necessary information. Explicit messages would have to be used to update the counts of effective and nominal parallelism, as well as the procedures each processor was executing. A further problem is that programs on these systems are often explicitly written to use a specific number of processors because of the need to explicitly control the communication pattern; removing one for sampling might require re-writing the program.

Alternative approaches also have significant drawbacks on such systems. In particular, recording and post-processing a complete trace may already be impractical for some programs, and will become more so as distributed memory multiprocessors support faster rates of interprocessor communication.

Efficient sampling could be implemented given hardware support for stopping all processors at close to the same time (i.e., by allowing a host computer to send parallel interrupts each processor). One of the reasons for using a dedicated processor is that interrupting any single processor to do sampling can distort the behavior being measured. If all were stopped together, the sample could be taken from that snapshot without measurement error. The sampling could be implemented efficiently by using the processing power of the stopped processors.

Absent hardware support, it may be possible to exploit the characteristics of parallel programs on distributed memory multiprocessors. Because of the requirement that interprocessor communication be explicitly programmed using messages, these systems are most commonly used for highly data-parallel applications with regular communication patterns. For these types of programs, at any point during the computation, each processor executes roughly the same section of code, although one may finish before another. As a result, sampling the behavior of each individual processor, and not the global state, may yield a sufficiently detailed picture of program performance.

The techniques used in Quartz also solve some problems with traditional approaches to tuning sequential program performance. One limitation to gprof is that it cannot be used to tune the implementation of operating system kernel-level routines, since interrupt-driven sampling cannot measure code that runs with interrupts disabled. By using a separate processor to do sampling, however, we would be able to accurately measure kernel-level execution. Note that by avoiding synchronization between the executing processors, or between them and the sampling processor, the processor in the kernel can execute the profiling code even if it holds the low level scheduler lock.

Similarly, by using a profile stack for sampling, we are able to account correctly for execution time spent out of the monitored address space. Because of the way gprof propagates execution time spent on behalf of a procedure, it cannot accurately attribute time spent executing in non-profiled code, or in the operating system on behalf of the program. However, a trend in the design of operating systems and large applications is to decompose them into separate modules in different address spaces, so that hardware protection mechanisms can be used for failure isolation. Much of the execution time of an editor, for instance, might be spent in another address space responsible for updating the display. The performance of the entire decomposed system could be measured by sampling profile stacks shared among all address spaces.

Data-oriented measurement can be helpful for tuning sequential programs as well as parallel programs. For some programs, knowing that a particular procedure accounts for a large proportion of the processing time may not be as useful as knowing that a particular object is expensive. For example, there is better resolution in a more finely tiled sphere, but it also costs more to draw. Gprof introduces a systematic bias in measuring object-oriented program behavior because it assumes that a procedure call always takes the same amount of time to execute (i.e., that the time does not depend on the data that is passed). Quartz avoids this bias by propagating execution times explicitly. While we currently make only limited measurements of data-oriented performance behavior, our design is extensible to a more thorough object-oriented implementation.

6. Conclusions

Achieving good performance from parallel applications is both crucial and challenging. We have discussed the design rationale, functionality, implementation, and use of Quartz, a tool for tuning parallel program performance on shared memory multiprocessors.

The philosophy underlying our work is that an effective tool for tuning parallel program performance must be based on a clear view of the causes of poor performance, and on a specific methodology for improving that performance. By being selective about what it measures and presents, Quartz can focus the programmer's attention on the information needed to tune performance. Measurement efficiency also results from designing the tool to record just the important behavior.

By correlating the execution of sections of code and the use of certain data objects with the concurrent behavior of other processors, Quartz assists in identifying areas of the program where re-structuring is necessary to improve performance, and in gaining insight into the types of re-structuring that will work. Because Quartz organizes performance information according to the logical structure of the program, the programmer can tune performance in a top-down fashion.

Acknowledgments

We would like to thank Hi-Keung Tony Ma for donating the application program we used for our case study, and Brian Bershad, Henry Levy, and John Zahorjan for their extensive comments.

References

[Anderson et al. 1989]
Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *1989 ACM SIGMETRICS and Performance '89 Conference on Measurement and Modeling of Computer Systems*, pp. 49-60, May 1989. To appear, *IEEE Transactions on Computers*, December 1989.

[Aral & Gertner 1988]
Ziya Aral and Ilya Gertner. Non-Intrusive and Interactive Profiling in Parasight. *Proc. ACM/SIGPLAN PPEALS 1988*, pp. 21-30.

[BBN 1985]
BBN Laboratories. Butterfly Parallel Processor Overview. 1985.

[Bershad et al. 1988]
Brian Bershad, Edward Lazowska, and Henry Levy. Presto: A System for Object-Oriented Parallel Programming. *Software: Practice and Experience 18*,8 (Aug. 1988), pp. 713-732.

[Burkhart & Millen 1989]
H. Burkhart and R. Millen. Performance Measurement Tools in a Multiprocessor Environment. *IEEE Transactions on Computers 38*,5 (May 1989), pp. 725-737.

[Carpenter 1987]
R.J. Carpenter. Performance Measurement Instrumentation for Multiprocessor Systems. In *High Performance Computer Systems*, ed. E. Gelenbe, North-Holland, pp. 81-92, 1987.

[Fowler et al. 1988]
Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey. An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors. *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[Graham et al. 1982]
S.L. Graham, P.B. Kessler, and M.K. McKusick. Gprof: A Call Graph Execution Profiler. *Proc. ACM SIGPLAN Symposium on Compiler Construction*, June 1982.

[Gupta 1989]
R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, April 1989, pp. 54-63.

[Halstead 1986]
R. Halstead, Jr. An Assessment of Multilisp: Lessons from Experience. *International Journal of Parallel Programming 15*,6 (Dec. 1986).

[Kerola & Schwetman 1987]
Teemu Kerola and Herb Schwetman. Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs. *Proc. 1987 ACM SIGMETRICS Conference*, May 1987.

[Ma et al. 1987]
H.T. Ma, S. Devadas, R. Wei, and A. Sangiovanni-Vincentelli. Logic Verification Algorithms and Their Parallel Implementation. *Proc. 24th Design Automation Conference*, pp. 283-290, July 1987.

[Malony et al. 1989]
Allen Malony, Daniel Reed, James Arendt, Ruth Aydt, Dominique Grabas, and Brian Totty. An Integrated Performance Data Collection, Analysis, and Visualization System. To appear, *Proc. 4th Conference on Hypercubes, Concurrent Computers, and Applications*, 1989.

[Miller & Yang 1987]
Barton P. Miller and C.-Q. Yang. IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs. *Proc. 7th International Conference on Distributed Computing Systems*, September 1987.

[Moeller-Nielsen & Staunstrup 1987]
P. Moeller-Nielsen and J. Staunstrup. Problem-Heap: A Paradigm for Multiprocessor Algorithms. *Parallel Computing 4*, North-Holland, 1987, pp. 63-74.

[Pfister et al. 1985]
G. Pfister, W. Brantley, D. George~~, S. Harvey~~, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weise. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. *Proc. 1985 International Conference on Parallel Processing.*

[Rodgers 1986]
David P. Rodgers. Personal communication.

[Segall & Rudolph 1985]
Zary Segall and Larry Rudolph. PIE: A Programming and Instrumentation Environment for Parallel Processing. *IEEE Software 2*,6 (November 1985).

[Sequent 1988]
Sequent Computer Systems, Inc. Symmetry Technical Summary.

[Thacker et al. 1988]
Charles Thacker, Lawrence Stewart, and Edward Satterthwaite Jr. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers 37*,8 (Aug. 1988), pp. 909-920.

[Yang & Miller 1988]
Cui-Qing Yang and Barton Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. *Proc. 9th International Conference on Distributed Computing Systems*, June, 1988, pp. 366-373.

Speedup Versus Efficiency in Parallel Systems

DEREK L. EAGER, JOHN ZAHORJAN, AND EDWARD D. LAZOWSKA

Abstract – If a software system can be structured as a collection of largely independent subtasks, significant reductions in elapsed time can be realized by executing these subtasks in parallel on multiple processors. This effect, known as *speedup*, typically increases (up to some limit) with the number of processors dedicated to the problem.

Along with an increase in speedup comes a decrease in *efficiency:* as more processors are devoted to the execution of a software system, the total amount of processor idle time can be expected to increase, due to factors such as contention, communication, and software structure.

This paper investigates the tradeoff between speedup and efficiency that is *inherent* to a software system. We show the extent to which this tradeoff is determined by the *average parallelism* of the software system, as contrasted to other, more detailed characterizations. We bound the extent to which both speedup and efficiency can simultaneously be poor: we show that for any software system and any number of processors, the sum of the average processor utilization (i.e., efficiency) and the attained fraction of the maximum possible speedup must exceed one. We give bounds on speedup and efficiency, and on the incremental benefit and cost of allocating additional processors. We give an explicit formulation, as well as bounds, for the location of the "knee" of the execution time – efficiency profile, where the benefit per unit cost is maximized.

Index Terms – Parallel software, performance, parallel computing, parallel software structure, bounds on performance, computer system performance analysis.

I. INTRODUCTION

Exploiting parallelism is an increasingly common approach to improving the performance of computer systems. In terms of hardware, this typically means providing multiple, simultaneously active processors. In terms of software, this typically means structuring a program as a set of largely independent subtasks.

In the sequential world the performance of a system usually can be adequately characterized in terms of the instruction rate of the single processor and the execution time requirement of the software on a processor of unit rate (which we refer to as its *service demand*). In the parallel world things are considerably more complex. In the hardware domain we must be concerned not only with the instruction rate of a processor, but also with factors such as the number of processors. In the software domain we must be concerned not only with service demands, but also with factors such as the structure of the software.

In evaluating a parallel system two performance measures of particular interest are *speedup* and *efficiency*. Speedup is defined for each number of processors n as the ratio of the elapsed time when executing a program on a single processor (the single processor *execution time*) to the execution time when n processors are available. In the notation that we shall use throughout this paper:

$$S(n) = \frac{T_1}{T_n}$$

Efficiency is defined as the average utilization of the n allocated processors. Ignoring I/O, the efficiency of a single processor system is 1. Speedup in this case is of course 1. In general, the relationship between efficiency and speedup is given by:

This work was supported by the National Science Foundation (Grants No. DCR-8302383, DCR-8352098, CCR-8619663 and CCR-8703049), the Natural Sciences and Engineering Research Council of Canada, the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

Derek L. Eager is with the Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1; this work took place while Eager was with the Department of Computational Science, University of Saskatchewan.

John Zahorjan and Edward D. Lazowska are with the Department of Computer Science, University of Washington, Seattle, WA 98195.

$$E(n) = \frac{S(n)}{n}$$

If efficiency remains at 1 as processors are added we have *linear* speedup. (Technically, for speedup to be linear requires only that $S(n) = \alpha n$ for some constant α, $0<\alpha\leq 1$, but we will use the stricter definition $\alpha=1$ throughout.) This is the ideal case, as improvements in speedup can be obtained at no cost in efficiency. Linear speedup is not achievable in general, because of contention for shared resources, the time required to communicate between processors and between processes, and the inability to structure the software so that an arbitrary number of processors can be kept usefully busy. Minsky and Papert noted evidence that the "typical" speedup has the form $S(n) = \log n$ [15]; other studies have provided evidence that much larger "typical" speedups can be attained [12].

Although the idea of speeding up computations through parallelism has existed for more than a century [11], general purpose systems based on multiple (five or more) processors have only recently become common (e.g., commercial machines by Sequent, Encore, Alliant, and BBN, and limited-edition machines such as IBM's RP3 and DEC's Firefly). The existence of such systems has stimulated widespread research activity: algorithms work concerned with parallel solutions in many problem domains, compiler work concerned with parallelizing code, architecture work concerned with how best to interconnect processors, etc. Obviously, results in these areas play a critical role in improving the speedup and efficiency properties of parallel systems.

This paper takes a more abstract view. Rather than studying specific implementations and implementation problems, we study the *tradeoff* between speedup and efficiency that is *inherent* to a software system. Further, we do not do this in the context of a specific software structure (e.g., as was done by Heidelberger and Trivedi [8] and by Fayolle, King and Mitrani [5]); instead, we derive relationships that can be very broadly applied. We are interested both in fundamental issues concerning the properties of this tradeoff, and in practical issues that might arise in considering specific software systems. Among the fundamental issues that we address are:

- To what extent is the speedup-efficiency tradeoff determined by the *average parallelism* of a software system, as contrasted to other, more detailed characterizations?
- How "bad" can speedup and efficiency *simultaneously* become?
- What is the nature of the "knee" of the execution time – efficiency profile, where the benefit (increase in speedup) per unit cost (decrease in efficiency) is maximized? For example, what guarantees can be made regarding speedup and efficiency values at the knee?

Among the practical issues related to specific software systems that we consider are:

- To achieve a given speedup, what efficiency penalty must be paid?
- What speedup advantage will result when increasing the number of processors by some factor, and what efficiency penalty will accompany it?
- What number of processors yields the knee of the execution time – efficiency profile?

Our objective is to address these questions by obtaining *bounds* on performance – bounds expressed in terms of the average parallelism measure of software structure. It should be clear that, given complete information regarding a specific software structure, precise answers (rather than bounds) could be obtained for many of these questions. There are two reasons, though, why bounds expressed in terms of one or a small number of parameters may be more desirable than precise solutions that require complete information:

- It is unlikely in practice that complete information will be available. For most software systems, parallelism will depend to some extent on the (unknown or varying) data that would be supplied as inputs. The volume of information required will in many cases be prohibitive.
- It often is the case that bounds based on simple characterizations yield more insight than exact answers utilizing complete information.

The results that we seek are in the spirit of the results of Chen [2], and, more widely known, in the spirit of Amdahl's law, which states that if a fraction f of a computation is inherently sequential, then the speedup $S(n)$ is bounded above by $\frac{1}{f + \frac{1-f}{n}}$ [1]. (Precisely, f is defined to be the ratio of the service demand of sequential parts of the computation to the service demand of the entire computation.) This is a simple upper bound on speedup that is expressed in terms of a single-parameter characterization of the software (f) and a single-parameter characterization of the hardware (n). It provides considerably more insight than more detailed alternatives, such as a table displaying exact speedup values computed for a number of specific software structures running on a number of specific hardware structures.

In II we describe our models of parallel software and of its execution, and the average parallelism measure that we use to characterize this software. Of special interest:

- We show that there are four equivalent definitions of average parallelism.
- We show that the number of available processors n and the average parallelism of the software structure A provide complementary *hardware* and *software* upper bounds on speedup.

In III we develop lower bounds on speedup and efficiency in terms of n and A, and apply these bounds to answer a number of the questions posed above. Specifically:

- We obtain lower bounds on the speedup and efficiency with n processors. These bounds apply to *any* work-conserving scheduling discipline, and are the best obtainable bounds. (Theorem 1.)
- Restricting ourselves to the processor sharing scheduling discipline, we show that, although tighter bounds can be obtained, they require information about the software structure in addition to the average parallelism. (Theorem 2.)
- We show that for any work-conserving scheduling discipline, any software structure, and any number of processors, the sum of the attained efficiency and the attained fraction of the maximum possible speedup must always exceed 1. In other words, we show that a low efficiency is guaranteed to "buy" a high relative speedup. (Corollary 1.1.)
- We bound efficiency in terms of the average parallelism and the speedup; in other words, we determine the efficiency penalty that must be paid to achieve some target speedup. (Corollary 1.2.)
- We show that average parallelism is a good characterization of the software structure. We do so in two ways. First, we show that a specific speedup estimate derived from only n and A can be in error by at most 33%. (Corollary 1.3.) Then, we show that only a slight improvement in our bounds can be achieved by using some additional information, specifically the fraction of work that is inherently sequential. (Corollaries 2.1 and 2.2.)

In IV we consider the incremental cost/benefit of adding processors. For an increase from n to kn processors, we obtain both upper and lower bounds on the increase in speedup and the decrease in efficiency. These bounds are expressed in terms of k, n, and A. (Theorem 3.)

Finally, in V we study questions concerning the knee of the execution time – efficiency profile.

- We show that this profile has a unique knee, and obtain an exact expression for the number of processors that attains this knee – an expression that requires complete information concerning the software structure (rather than simply the average parallelism). (Theorem 4.)
- We show that when this number of processors is allocated, the attained speedup is *at least* 50% of the maximum possible and the efficiency is *at least* 50%. (Theorem 5.)
- We obtain a bound for this "optimal" number of processors in terms of the average parallelism, A. (Theorem 6.)
- We show that when the number of available processors is equal to the average parallelism, the guarantees regarding speedup and efficiency are identical to those at the knee. (Theorem 7.)

II. THE SYSTEM MODEL AND THE AVERAGE PARALLELISM MEASURE

In A we outline the graph model of parallel software that we will use, as well as our model of execution. The former model reflects our goal of fundamental and generally applicable insights, and incorporates only those aspects common to all parallel software systems. The latter model reflects, in addition, our focus on the parallelism inherent to a software system, and makes an assumption of "ideal hardware". In B we describe the average parallelism measure that we use to characterize software parallelism.

A. The System Model

We represent the software component of the system using a traditional graph model (e.g., [6]). In this model a software system is represented by an acyclic directed graph. Each vertex of the graph corresponds to a "subtask" of the software system. Each subtask has an associated processor service demand. Precedence constraints may exist among the subtasks; for example, the initiation of a subtask may require data that is available only after the termination of some other subtask. These precedence constraints are modelled by the arcs of the graph: an arc from vertex A to vertex B means that subtask B cannot begin execution until subtask A completes execution. (It would be incorrect to think that a subtask in this model necessarily corresponds directly to a process or to some other operating system or programming language construct. Rather, a subtask corresponds to an "independent unit of sequential work". A single process might be represented by several such units, for example, with the precedence constraints representing synchronization requirements achieved by some communication primative.) Fig. 1 illustrates a software structure graph that might arise from an algorithm such as Quicksort; service demands appear within the vertices.

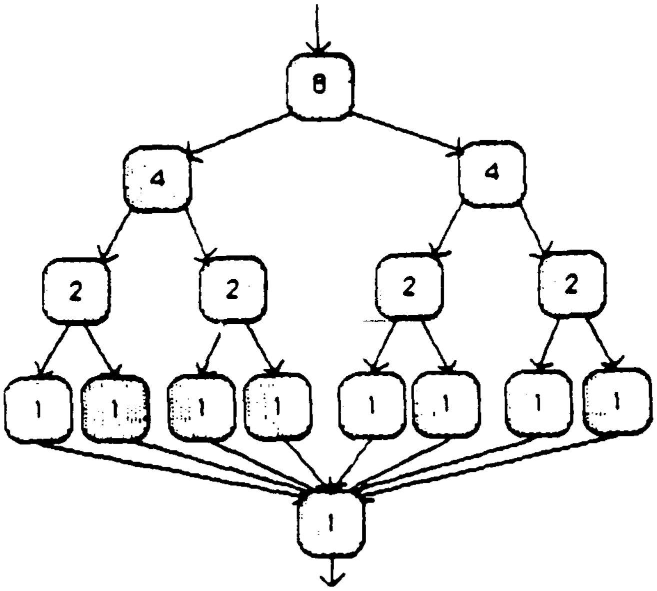


Fig. 1. Graph representation of an example software system.

The hardware component of the system is modelled as some number n of identical processors, each of unit speed. We assume that n is constant throughout the execution. In practice, n might be determined by either hardware or software. If the entire computer is devoted to a single task, the number of available processors is fixed by the hardware. If the computer is multiprogrammed, the operating system may choose to allocate a fixed subset of the processors to each task during its execution. Some of the questions that we consider in this paper are more relevant in one of these contexts than in the other. For example, questions regarding the knee of the execution time – efficiency profile are most relevant to the multiprogramming context, in which there is considerable freedom in determining the number of processors allocated to each software system.

It is possible that at various points during execution the number of runnable subtasks will exceed the number of available processors. A scheduling algorithm then is required to decide which subtasks should be run. Some of our results will be established for any scheduling discipline that is work-conserving [9]. (A work-conserving discipline is one that never leaves idle a subtask that is eligible for execution when there is a processor available.) Other results will consider a specific discipline, *processor sharing*. Under this discipline, if k subtasks are eligible for execution and there are n available processors ($n < k$), each subtask receives service at a rate that is $\frac{n}{k}$ times the rate at which it would receive service if a processor were dedicated to it.

In closing this section, we should specifically discuss the representation of overheads in our execution model. The principal focus of this paper is the influence of software structure on parallel program performance. Overheads such as those due to interconnection network topologies, memory contention, and locking are of course another important influence on performance, but are not our principal focus. These overheads are represented by including them in the service demands of the various subtasks in the graph. This is a common approach in computer system performance analysis. The implicit assumption is that these overheads are *fixed* – that they do not vary with the number of processors dedicated to the computation nor with the schedule used.

B. *The Average Parallelism Measure*

The graph representation of a parallel software system contains complete information about the parallelism inherent in that system. We argued in I that this representation is too detailed to be practical or yield insight. We seek a simpler characterization that still captures the essential behavior of the software.

An example of such a characterization is the one used by Amdahl [1]: the fraction f of a computation that is inherently sequential. For our work, we have chosen to use a different, fairly common (e.g., [7]), and intuitively appealing measure: *average parallelism*. Average parallelism can be rigorously defined in four equivalent ways:

A. the average number of processors that are busy during the execution time of the software system in question, given an unbounded number of available processors;

B. the speedup, given an unbounded number of available processors;

C. the ratio of the total service required by the computation (the sum of the service demands of the subtasks) to the length of a longest path in the subtask graph (the length of a path is the sum of the service demands of its subtasks); and

D. the intersection point of the *hardware bound* and the *software bound* on speedup (these will be defined shortly).

The equivalence of these four definitions is not entirely obvious. Recall that speedup with n processors, $S(n)$, is defined as the ratio of the execution time when only a single processor is available to the execution time when n processors are available. Since the former is equal to the total service demand, and the ratio of the total service demand to the execution time gives the average number of busy processors, definition (B) is equivalent to definition (A).

If an unbounded number of processors is available, the execution time of a software system is simply the total service demand along some longest path. Hence, from the definition of speedup, definition (C) is equivalent to definition (B).

There are two simple upper bounds on speedup. The *hardware bound* reflects the limitation imposed by the hardware, and is given by the number n of available processors. This bound can be achieved only if all n processors can be kept busy all of the time. The *software bound* reflects the limitation imposed by the software, and is derived by noting that, no matter how many processors are available to a system, the execution time must be at least as long as the length of a longest path. Hence, the speedup is at most the ratio of the total service demand to the length of a longest path. The hardware and software bounds, and the actual speedup function, are depicted in Fig. 2 for the example software system whose directed graph representation is shown in Fig. 1.

The intersection point of the hardware and software bounds on speedup is significant: when additional processors are allocated, it is certain that there is not enough parallelism in the software to keep all of the processors busy all of the time. This intersection point is the point where n (the hardware bound) is identical to the ratio of the total service demand to the length of the longest path (the software bound). Thus, definition (D) is equivalent to definition (C), and all four definitions are now shown to be equivalent.

We note that the hardware and software bounds on speedup are analogous to (and have an identical form as) the *Asymptotic Bound Analysis* (*ABA*) bounds on system throughput in a queueing network model of a computer system in which a number of identical, independent processes compete for service at a collection of system resources [4, 16]. There is a simple mapping between the two problem domains, with the number of independent processes in the ABA model corresponding to the number of processors in our model, and the bottleneck service demand in the ABA model corresponding to the length of a longest path in the directed graph representing the software system in our model. A similar mapping was employed by Kumar and Gonsalves [13] for performance models of software containing critical sections.

It is natural to ask how to determine the average parallelism of a particular software system. There are analytic approaches (considering the graph representation of the system) and experimental approaches (running the software with a sufficient number of processors). As with Amdahl's simple measure, though, the important issue is not how the measure is determined, but rather that once it has been determined, it provides a succinct characterization of the inherent parallelism of the software system. As the remainder of this paper will show, there is a considerable amount of information built into this measure.

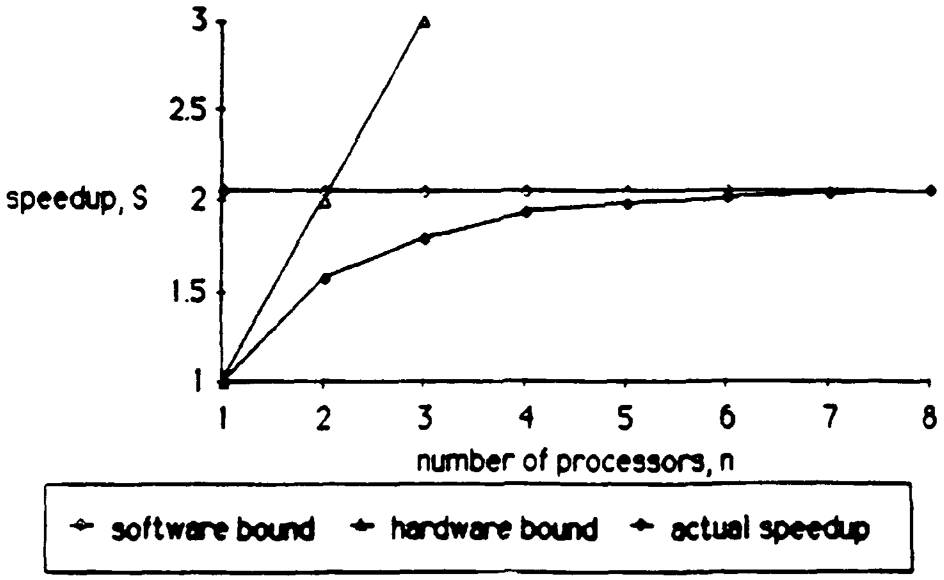


Fig. 2. Upper bounds and actual speedup for the graph in Fig. 1.

III. LOWER BOUNDS ON SPEEDUP AND EFFICIENCY, AND APPLICATIONS OF THESE BOUNDS

In II we showed the upper bounds on speedup that are established by hardware and by software. There are corresponding upper bounds on efficiency, since the average processor utilization (efficiency) can be computed as the speedup divided by the number of available processors.

In this section we derive lower bounds on speedup and efficiency, and apply these bounds to a number of questions regarding the speedup-efficiency tradeoff.

A. Lower Bounds on Speedup and Efficiency

Theorem 1: Let A denote the average parallelism of a software structure, $S(n)$ the speedup with n processors, and $E(n)$ the efficiency with n processors. For any work-conserving scheduling discipline

$$S(n) \geq \frac{nA}{n+A-1}$$

and

$$E(n) \geq \frac{A}{n+A-1}$$

These bounds can be attained.

Proof: Because speedup and efficiency are related in a simple way, it suffices to show the speedup result. Consider an arbitrary software structure with average parallelism A. Let T_∞ denote the elapsed time of its execution given an unlimited number of processors. It follows from the definition of average parallelism that the total processor busy time accumulated during such an execution (summed over all n processors) is $T_\infty A$.

Now, suppose that this software structure is executed using n processors and some arbitrary work-conserving scheduling discipline. Since the discipline is work-conserving, it follows that the total busy time (summed over all processors) is, as before, $T_\infty A$. The execution time is then given by $\frac{T_\infty A + I(n)}{n}$, where $I(n)$ denotes the total idle time (summed over all processors) that is accumulated during this execution. From the definition of speedup, noting that the sequential execution time is $T_\infty A$, we then have $S(n) = \frac{nA}{A + \frac{I(n)}{T_\infty}}$.

To establish the desired result, we only need to show that $I(n) \leq T_\infty(n-1)$. To this end, define $\omega(t)$ to be the portion of the original software structure graph that has not completed execution at time t. $\omega(t)$ includes those tasks that have not yet been initiated, and those tasks that have been initiated but not yet completed. The service demand of each task in $\omega(t)$ is its original service demand diminished by the amount of service (if any) already provided to the task. (The precedence arcs in $\omega(t)$ are identical to those in the original graph.)

Define $L(t)$ to be the length (i.e., the total service demand) of a longest path within $\omega(t)$. Note that the value of $L(t)$ varies from T_∞ (at the start of the execution) to zero (at the end of the execution). If, at some time t, $L(t)$ is *not* decreasing, there must be some task at the head of a longest path in $\omega(t)$ that is not being executed. Since no precedence constraints prevent the execution of such a task, and since the scheduling discipline is work-conserving, it must be the case that there are no idle processors at time t. Thus, processors can be idle only during those periods of time when $L(t)$ is decreasing. Since $L(t)$ decreases (linearly) for a total length of time T_∞, and since at most $n-1$ processors can be idle at any point in time, the total idle time $I(n)$ is at most $T_\infty(n-1)$, which establishes the result.

We show that the speedup bound can be attained by means of an example. Consider First-Come-First-Served scheduling, and a software system that consists of a subtask that, upon completion, enables $kn+1$ additional subtasks, where kn is some multiple of n that is greater than A. If we constrain the service times of each of the $kn+1$ additional subtasks to be identical, the service time of the first subtask can be derived as $T_\infty \frac{kn+1-A}{kn}$, and that of each of the remaining subtasks as $T_\infty \frac{A-1}{kn}$. The speedup is then given by

$$S(n) = \frac{(kn+1)T_\infty \frac{A-1}{kn} + T_\infty \frac{kn+1-A}{kn}}{T_\infty \frac{kn+1-A}{kn} + (k+1)T_\infty \frac{A-1}{kn}}$$

This reduces to the speedup given in the theorem.

QED

Note that if $n \ll A$, $S(n) \rightarrow n$, and if $n \gg A$, $S(n) \rightarrow A$. Fig. 3 adds the lower bound of this theorem to the upper bounds displayed in Fig. 2. Bear in mind that the bounds of this theorem apply to *any* work-conserving scheduling discipline. No matter how poorly designed such a discipline may be, or how baroque a software structure is presented, the behavior of the software system can be no worse than the stated bounds. Furthermore, these are the best such bounds obtainable, since for any choice of A and n there exists a choice of work-conserving discipline and software structure such that the system performance is no better than that given by the bounds.

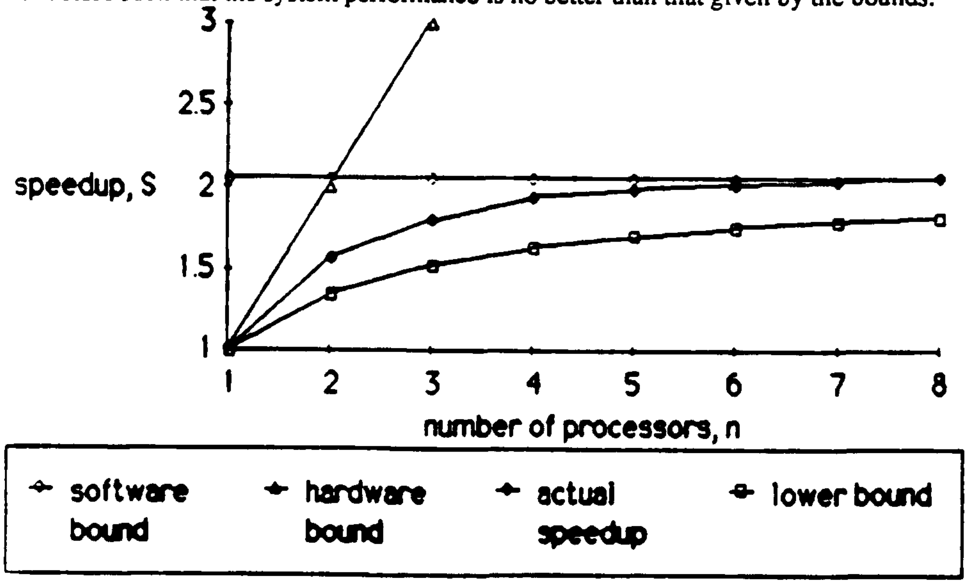


Fig. 3. Lower bound added to Fig. 2.

Because Theorem 1 is concerned with the worst case over the entire space of work-conserving disciplines, it is possible that the bound does not give a reasonable indication of the performance to be expected of "rational" scheduling disciplines. Theorem 2 specializes the results to the case of processor sharing scheduling, an idealization of round robin scheduling. We find that, although a tighter bound can in fact be obtained, this requires not only that we consider a specific scheduling discipline, but also that we provide additional information about the software structure, specifically, the maximum parallelism.

Theorem 2: Let A denote the average parallelism, $m_{\max}$ the maximum parallelism (the maximum number of processors that are simultaneously busy when an unlimited number are available), $S(n)$ the speedup with n processors, and $E(n)$ the efficiency with n processors. With processor sharing scheduling

$$S(n) \ge \min\left[A\ ,\ \frac{nA}{n+A-1-\dfrac{(n-1)(A-1)}{m_{\max}-1}}\right]$$

and

$$E(n) \ge \min\left[\frac{A}{n}\ ,\ \frac{A}{n+A-1-\dfrac{(n-1)(A-1)}{m_{\max}-1}}\right]$$

These bounds can be attained. (The min function is necessary only to take care of the extreme case in which $n > m_{\max}$.)

Proof: Consult the Appendix.

A key point is that if we have no information about the maximum parallelism, i.e., if the maximum parallelism can be arbitrarily high, then these bounds reduce to those given in Theorem 1.

B. Applications

1) How "Bad" Can Speedup and Efficiency Simultaneously Become?

Typically, as we make additional processors available to a software system, increases in speedup are obtained at the expense of decreases in efficiency. It is natural to wonder if a low efficiency is guaranteed to "buy" a high speedup, or whether a poor choice of scheduling discipline and/or an inappropriate number of available processors might result in both low efficiency and low speedup. Corollary 1.1 offers reassurance in this regard.

Corollary 1.1: For any work-conserving scheduling discipline, any software structure, and any number of processors, the sum of the attained efficiency and the attained fraction of the maximum possible speedup must always exceed 1, i.e., $E(n) + \frac{S(n)}{A} > 1$.

Proof: From Theorem 1, $\frac{S(n)}{A} \ge \frac{n}{n+A-1}$ and $E(n) \ge \frac{A}{n+A-1}$. Thus, $\frac{S(n)}{A} + E(n) \ge \frac{n+A}{n+A-1} > 1$.

QED

Thus, an average processor utilization (efficiency) of 20%, for example, implies an attained speedup of more than 80% of the maximum possible.

2) To Achieve a Given Speedup, What Efficiency Penalty Must be Paid?

Here, we wish to obtain bounds on efficiency, given that sufficient processors have been made available to attain some target speedup S. (Clearly, S can be at most the maximum possible speedup, as given by the average parallelism A.)

Corollary 1.2: For any non-sequential program structure (that is, any structure with $A > 1$) and any work-conserving scheduling discipline, $E(n) \ge \frac{A-S(n)}{A-1}$.

Proof: From Theorem 1, $S(n) \ge \frac{nA}{n+A-1}$, so $n \le \frac{S(n)(A-1)}{A-S(n)}$, and thus $\frac{S(n)}{n} = E(n) \ge \frac{A-S(n)}{A-1}$.

QED

Note that for small values of S, the efficiency penalty is guaranteed to be small. However, as S approaches the maximum possible speedup A, the efficiency may, in the worst case, degrade linearly to arbitrarily low values. This agrees with our intuition regarding the likely effect of trying to exploit all of the possible parallelism in a system by dedicating large numbers of processors.

3) To What Extent is the Speedup-Efficiency Tradeoff Determined by Average Parallelism?

We address this issue by answering two related questions. First, how tightly can speedup be bounded using only the average parallelism of a software system and the number of available processors? Second, how much additional information regarding speedup is provided by knowledge of (i) the maximum parallelism, or (ii) the fraction of the total work that is inherently sequential? We consider only speedup, since efficiency is easily derived once the speedup is known.

Fig. 3 illustrated, for a specific software structure, the actual speedup function $S(n)$, the upper bounds established by software (A) and hardware (n), and the lower bound of Theorem 1 ($\frac{nA}{n+A-1}$). We will measure the "tightness" of these bounds in the general case by determining the maximum possible error in an estimate of the speedup function that is computed by a particular averaging of the bounds.

Corollary 1.3: For any work-conserving scheduling discipline, the speedup estimate

$$\hat{S}(n) = 2 \times \frac{\min(n, A)\frac{nA}{n+A-1}}{\min(n, A) + \frac{nA}{n+A-1}}$$

has a relative error of less than 33%, i.e., $\frac{|\hat{S}(n) - S(n)|}{S(n)} \leq 0.33$.

Proof: It is straightforward to show that the relative error in this estimate is maximized when the true speedup attains either the lower or the upper speedup bound. Therefore, the relative error is at most

$$\frac{\min(n, A) - \frac{nA}{n+A-1}}{\min(n, A) + \frac{nA}{n+A-1}}$$

which is maximized at $n = A$ and is thus strictly less than 33%.

QED

Thus, knowing only the average parallelism and the number of available processors, a speedup estimate can be computed that is guaranteed to have a relative error of less than 33%.

This result indicates that at most a modest benefit can be had by considering measures of software parallelism more detailed than average parallelism. Nonetheless, we will briefly examine two such measures that have a reasonable likelihood of being known or reliably estimated in practice: the maximum parallelism and the fraction of the total work that is inherently sequential. (Another reasonable characteristic to consider might be the fraction of the total work that is accomplished with the (known) maximum parallelism. The additional benefit of this information is, however, very similar in nature to that of the fraction of the total work that is inherently sequential, and we do not treat it explicitly here. We consider other measures, such as the variance in parallelism or the percentiles of parallelism, to be too detailed for practical use.)

Knowledge of the maximum parallelism can be used to tighten only the lower bound on speedup. The extent of this improvement is illustrated in Theorem 2 for the processor sharing scheduling discipline. If $m_{\max}$ (the maximum parallelism) is large, the benefit of knowing its value is minimal – one might just as well assume that it is unbounded. If $m_{\max}$ is small, the speedup bound is considerably tightened. (Of course, $m_{\max}$ must be at least as large as A.) For intermediate values, the benefit of knowing $m_{\max}$ is maximized when n is close to A, which corresponds to the region of greatest uncertainty in speedup when only the average parallelism is known. For example, suppose that $n = A$ and that $m_{\max} = kA$ for some integer k. Then knowledge of $m_{\max}$ tightens (increases) the speedup lower bound by a factor of approximately $\frac{2k}{2k-1}$.

Knowledge of the fraction f of work that is inherently sequential (recall that this is defined to be the ratio of the service demand of the sequential parts of the computation to the total service demand of the computation) can be used to tighten both the upper and the lower bounds on speedup. The extent of these improvements will be illustrated for the processor sharing scheduling discipline, using the following two corollaries to Theorem 2. We first consider the improvement in the lower bound.

Corollary 2.1: Let A denote the average parallelism, f the fraction of work that is inherently sequential, and $S(n)$ the speedup with n processors. With processor sharing scheduling and $n \geq 2$

$$S(n) \geq \frac{nA}{n+A-1-(1-fA)}$$

(fA will always be between 0 and 1.)

Proof: Consult the Appendix.

Comparing this bound to that given in Theorem 1 (or, equivalently, to that given in Theorem 2 when $m_{\max}$ is unknown), we note that for reasonably large n or A, knowledge of f provides negligible improvement in the speedup lower bound.

We next consider the improvement in the upper bound presented in II, $\min(n, A)$.

Corollary 2.2: Let A denote the average parallelism, f the fraction of work that is inherently sequential, and $S(n)$ the speedup with n processors. With processor sharing scheduling

$$S(n) \leq \min\left(\frac{n}{1+f(n-1)}, A\right)$$

Proof: Consult the Appendix.

(Note that the first term of this bound is identical to the bound given by Amdahl.)

Comparison with the bound from II shows that for small f, the improvement is negligible. As f approaches 1 the improvement increases, and, in fact, the new upper bound approaches the lower bound given by Theorem 1. For fixed f and A, the improvement is maximized when n is closest to A, which corresponds to the region of greatest uncertainty in speedup when only the average parallelism is known.

In summary, the average parallelism of a software system does, to a considerable extent, determine the associated speedup-efficiency tradeoff. Knowing only the average parallelism and the number of processors, a speedup estimate can be derived that has a relative error of less than 33%. Knowledge of other system characteristics such as the maximum parallelism $m_{\max}$ or the fraction f of work that is inherently sequential is of limited benefit: these measures yield useful information only when they indicate a severe constraint on parallelism, for example, a large value of f or a small value of $m_{\max}$.

IV. INCREMENTAL COST AND BENEFIT OF ADDING PROCESSORS

In this section we study the cost (decreased efficiency) and benefit (increased speedup) that will result when increasing by some factor the number of processors allocated to a software system. Theorem 3 addresses these questions by providing bounds expressed in terms of average parallelism.

Theorem 3: With processor sharing scheduling, an increase in the number of processors from n to kn $(n, k \geq 1)$ affects speedup as follows:

$$\max\left(1, \frac{kA}{(k-1)n+A}\right) \leq \frac{S(kn)}{S(n)} \leq \min\left(1+(A-1)\frac{k-1}{kn-1}, k\right)$$

Correspondingly, efficiency is affected as follows:

$$\max(\frac{1}{k}, \frac{A}{(k-1)n+A}) \le \frac{E(kn)}{E(n)} \le \min(\frac{1}{k} + (A-1)\frac{1-\frac{1}{k}}{kn-1}, 1)$$

These bounds can be attained.

Proof: Consult the Appendix.

A number of observations can be made regarding these bounds. We first show that they are consistent with earlier bounds, and then consider the insight that they provide regarding system behavior. Since the efficiency bounds are so closely related to the speedup bounds, we discuss only the latter directly.

For $n=1$, Theorem 3 provides bounds on speedup with k processors, since the factor by which speedup increases is equal to the speedup itself. These bounds correspond exactly to those given in II (the upper bounds established by hardware and by software) and in III (the lower bound of Theorem 1). Also, note that when $k \to \infty$ (with fixed n, A), the resulting speedup must be the maximum possible speedup, A, and that therefore the bounds on the change in speedup can be used to bound the original speedup with n processors. The upper bound on the change in speedup reduces to $\frac{n+A-1}{n}$ in this case, and the lower bound reduces to $\max(1, \frac{A}{n})$. Since A is the resulting speedup, the original speedup with n-processors is bounded below by $\frac{nA}{n+A-1}$ and above by $\min(n, A)$, again corresponding exactly to the bounds given in II and III.

Consider now the system behavior as the number of processors is increased, for various initial numbers of processors. For n and kn that are small relative to A, the lower bound on the factor by which speedup increases guarantees a speedup close to linear in the number of processors. For example, with $n=\frac{A}{9}$, doubling the number of processors will cause an increase in speedup of at least 80%. As n increases beyond A, the upper bound on the factor by which speedup may increase approaches the lower bound of 1 quickly. For example, suppose that initially we use A processors, and that A is large. If we then double the number of processors from A to $2A$, speedup will increase by at most 50%. If we double the number of processors again (from $2A$ to $4A$), speedup will increase by at most 25%. In general, at the i-th doubling, speedup will increase by at most $\frac{1}{2i} \times 100\%$.

As we have seen several times before, the greatest uncertainty arises when the number of processors is close to the average parallelism A. For example, with $n=\frac{2}{3}A$ (and large A), doubling the number of processors could increase speedup by anywhere from 20% to 75%.

V. THE KNEE OF THE EXECUTION TIME – EFFICIENCY PROFILE

Profiles that plot a measure of "benefit" against a measure of "cost" arise naturally in many areas; for example, the throughput-delay profile in computer-communication system design [10] and the lifetime curve in memory management [3]. The concept of the *knee* of such a profile [3] (or, alternatively, the point of "maximum power" [10]) is a fundamental one. The knee is the point where the benefit per unit cost is maximized, and, intuitively, represents an optimal system operating point.

In this section we investigate an important cost-benefit tradeoff in parallel systems: the *execution time – efficiency profile*. There are two equivalent views that motivate this tradeoff. In the first of these, we view efficiency as an indication of benefit (the higher the efficiency, the higher the benefit), and execution time as an indication of cost (the higher the execution time, the higher the cost). The implied system objective is to achieve efficient usage of each processor, while taking into account the cost to users in the form of increased task execution times. Since efficiency is closely related to the "per-processor throughput", this view is analogous to that motivating the throughput-delay profile in computer-communication network design.

The second, equivalent view has a somewhat different implied objective. Here, execution time is taken as an indication of benefit (the lower the execution time, the higher the benefit), and efficiency is taken as an indication of cost (the lower the efficiency, the higher the cost). The implied objective is to achieve low task execution times, while taking into account the "opportunity cost" of low efficiency. (In a multiprogramming environment, a low efficiency implies that processors could have been more appropriately allocated to a different task.)

The execution time – efficiency profile, motivated by each of these views, is a graph in which execution time is given on the y-axis and efficiency on the x-axis. Each point represents the combination of execution time and efficiency achieved by some particular number of processors. The knee of the profile occurs where the ratio of efficiency to execution time, $\frac{E(n)}{T_n}$ (T_n is the execution time when n processors are allocated), is maximized. As an example, Fig. 4 shows the profile for the software system whose graph representation and speedup function are given in Figs. 1 and 2, respectively. The knee is indicated by the arrow.

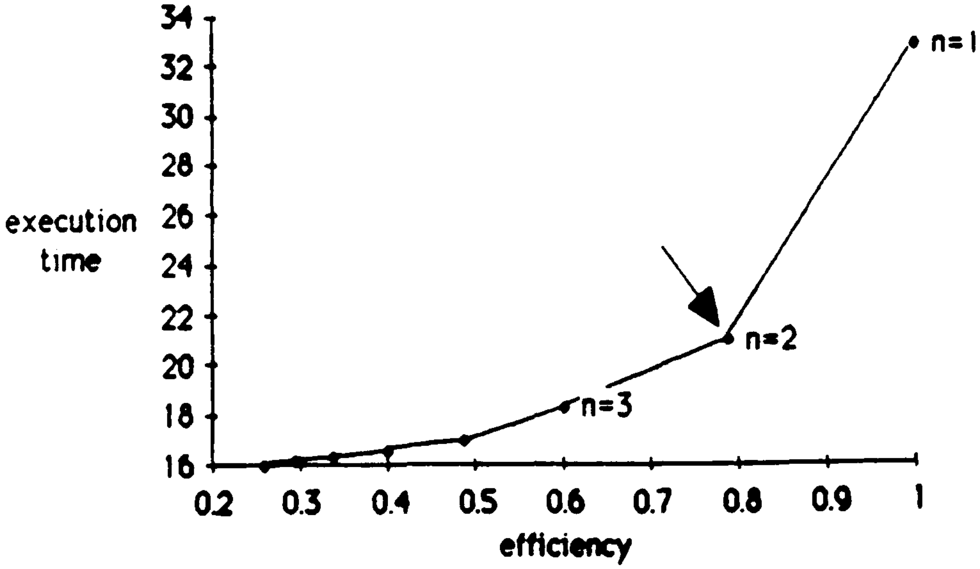


Fig. 4. Execution time – efficiency profile corresponding to Fig. 1.

It is important to be able to determine the number of processors that yields the knee. In a multiprogramming environment, for example, this would represent the appropriate allocation of processors to each job.

In A we show how to find the location of the knee for an arbitrary software system, assuming complete information about that system. Previously, the location of the knee had been found efficiently only for very regular structures (e.g., [14]). Using our new formulation for the location of the knee, we show several properties regarding speedup and efficiency that will hold if the number of processors allocated is the number that attains the knee.

The approach in A requires complete information. In B we bound the location of the knee in terms of average parallelism. Interestingly, the average parallelism itself is essentially at the midpoint of these bounds. Thus, choosing to allocate a number of processors equal to the average parallelism measure is appropriate, in the sense that it yields a point on the execution time – efficiency profile relatively close to the knee. We show several properties regarding speedup and efficiency that will hold if the number of processors allocated is identical to the average parallelism A, and compare these properties with those that hold at the knee.

A. The Exact Location of the Knee

In Theorem 4, we show that the execution time – efficiency profile for any software system must have a knee (a point where the ratio of efficiency to execution time is maximized), we show that this knee must be unique, and we give the location of the knee. Here, as in all of V, we assume processor sharing scheduling, and allow a nonintegral number of processors n. (Results that allow a nonintegral number of processors are in some sense "more general" than those for an integral number of processors. The nonintegral results can easily be specialized to integers. A nonintegral number of processors can be viewed as resulting from the sharing of a processor between two jobs in a multiprogrammed parallel environment.)

Theorem 4: Let p_m denote the proportion of time that m processors are simultaneously busy when an unlimited number are available, and let m_{max} denote the maximum parallelism (the maximum number of processors that are simultaneously busy when an unlimited number are available). Under the processor sharing discipline, the execution time – efficiency profile for any software system has a unique knee. The number of processors that attains the knee, i.e., that maximizes $\frac{E(n)}{T_n}$, is given by

$$n = \frac{\sum_{m=\lfloor n \rfloor+1}^{m_{\max}} p_m m}{\sum_{m=1}^{\lfloor n \rfloor} p_m}$$

if this equation has a solution, and by the unique integer n satisfying

$$\frac{\sum_{m=n+1}^{m_{\max}} p_m m}{\sum_{m=1}^{n} p_m} \le n \le \frac{\sum_{m=n}^{m_{\max}} p_m m}{\sum_{m=1}^{n-1} p_m}$$

otherwise.

Proof: Consult the Appendix.

As noted earlier, the knee of the execution time – efficiency profile intuitively represents a good system operating point. We now consider ~~what guara~~ntees are possible. Measures of interest include the *attained speedup* $S(n)$ in comparison with the maximum achievable speedup $S(m_{\max})$, the *efficiency* $E(n)$, the *utilization of the k-th last processor* for various values of k, and the *utilization of the k-th additional processor* for various values of k. These last two quantities require some explanation (during which, for simplicity, we will assume an integral number of processors).

The utilization of the k-th last processor indicates whether or not the number n of available processors is too large, and is defined as follows. Processors are numbered from 1 to n. If the number of subtasks s that are eligible for execution at a particular instant is less than n, then only processors 1 to s are utilized (even if physically this would require preempting a subtask executing on a higher numbered processor and re-scheduling it on a lower numbered processor). Given this dispatching rule, the utilization of the k-th last processor is defined as the average utilization of processor $n-k+1$. Note that all processors numbered from 1 to $n-k$ must have utilizations greater than or equal to this value, and that all processors numbered from $n-k+2$ to n must have utilizations that are less than or equal to this value.

The utilization of the k-th additional processor indicates whether or not the number n of available processors is large enough. As above, processors are numbered from 1 to n. Using the dispatching rule just described, the utilization of the k-th additional processor is defined as the utilization of the $n+k$-th processor when the number of available processors is increased to $n+k$. Note that this utilization will be less than or equal to that of all other processors, in particular processors $n+1$ to $n+k-1$.

Let the number of processors that yields the knee be denoted by K. In Theorem 5 we give bounds on the attained speedup in comparison with the maximum possible speedup, the efficiency, the utilization of the k-th last processor, and the utilization of the k-th additional processor, for a number of available processors equal to K. For these last two quantities we fix k to be 1 – we consider only the utilization of the last processor and the utilization of a single additional processor.

Theorem 5: Under the processor sharing discipline, when the number of available processors is equal to K (achieving the knee of the execution time – efficiency profile), the attained speedup is *at least* 50% of the maximum possible, the efficiency is *at least* 50%, the utilization of the last processor is *at least* 50%, and the utilization of a single additional processor is *no more than* 50%. These bounds can be attained in the limit as $A \to \infty$.

Proof: Consult the Appendix.

Intuitively, Theorem 5 shows that the conflicting goals of high efficiency and large speedup are being "perfectly balanced" when we choose to operate at the knee of the execution time – efficiency profile. An efficiency of at least 50% is guaranteed, as is a speedup of at least 50% of the maximum possible. All of the current processors are utilized at least 50%, but if another processor were introduced it would be utilized no more than 50%.

B. Bounds on the Location of the Knee

The results of the previous section allow the location of the knee to be determined precisely, but require complete information concerning the software system (in particular, the proportions p_m). We now bound the location of the knee in terms of average parallelism. These bounds indicate the close correspondence between the average parallelism A and the knee location K.

Theorem 6: Under the processor sharing discipline, the number of processors K that yields the knee of the execution time – efficiency profile must satisfy

$$\frac{A}{2} \le K \le 2A-1$$

When integers, these bounds on K can be attained.

Proof: Consult the Appendix.

Theorem 6 shows that the number of processors that yields the knee is at most a factor of two different from the average parallelism. This suggests that it might be reasonable to use the average parallelism itself as an approximation to the knee location. The reasonableness of this approximation depends on how the properties given in Theorem 5 are affected by using A rather than K processors. This concern is addressed in Theorem 7, where for clarity we consider only integral A.

Theorem 7: Under the processor sharing discipline, when the number of available processors is chosen to equal the average parallelism A, the attained speedup is *at least* 50% of the maximum possible, the efficiency is *at least* 50%, the utilization of the k-th last processor ($k<A$) is *at least* $\frac{k}{k+A}$, and the utilization of the k-th additional processor is *no more than* $\frac{A-1}{A+k-1}$. These bounds can be attained.

Proof: Consult the Appendix.

Theorem 7 shows that when the number of available processors is equal to the average parallelism, the guarantees regarding speedup and efficiency are *identical* to those at the knee – speedup and efficiency each must be within 50% of their maximum values. However, in contrast to the situation at the knee, there are no guarantees as to whether the number of processors is actually somewhat too large (implying that nearly the same speedup could be achieved with fewer processors), or somewhat too small (implying that increases in speedup could be achieved with a slight increase in the number of processors).

VI. CONCLUSIONS

In 1967 Amdahl gave a simple bound on the speedup that could be obtained by parallel processing as a function of the fraction of sequential code in a computation. This bound has proven useful in shaping our understanding of parallel systems because it strikes a useful balance between simplicity and precision.

In this paper we have investigated the tradeoff between execution time speedup and processor efficiency that arises from the inherent characteristics of parallel software systems. Like Amdahl, our goal is to find bounds on performance that can be expressed as functions of simple measures of the parallel system, and that provide insight into the behavior of these systems. Using average parallelism as our characterization, we have shown that speedup and efficiency cannot simultaneously be low, regardless of scheduling discipline or software structure. This indicates that parallel software systems are "robust", in the sense that very poor, anomalous behavior cannot exist. We have also shown that average parallelism provides a great deal of information about the system, in the sense that incrementally more information allows one to tighten the performance bounds only marginally.

Finally, we have examined questions related to the construction and management of parallel systems. Our results bound the efficiency cost and speedup benefit possible by altering the number of allocated processors. The knee of the execution time – efficiency profile has been investigated, a concept with application to multiprogrammed multiprocessors with static processor allocation. We use an explicit formulation of the knee location to show that this location is well approximated by the average parallelism, and to derive bounds on the speedup and efficiency values that are attained at the knee.

It is clear that for any particular problem, all measures of interest can be computed exactly (within our model) from the full specification of the software structure graph. Our intention has been to show formally that a vastly simpler characterization of the software provides considerable information about how the system will behave, and to use this characterization to develop simple relationships that enhance our understanding of the tradeoffs inherent in the design and use of parallel systems.

ACKNOWLEDGEMENT

Jean-Loup Baer, Sue Owicki, Ken Sevcik, and two anonymous referees offered helpful comments on earlier versions of this paper.

REFERENCES

[1] G.M. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," in *Proc. AFIPS Vol. 30*, pp. 483-485 1967.

[2] T.C. Chen, "Overlap and pipeline processing," in H. Stone, ed., *Introduction to Computer Architecture*, SRA, pp. 375-431, 1975.

[3] P.J. Denning, "Working sets past and present," *IEEE Trans. on Software Engineering SE-6*,1, pp. 64-84, 1980.

[4] P.J. Denning and J.P. Buzen, "The operational analysis of queueing network models," *Computing Surveys 10*,3 pp. 225-261, 1978.

[5] G. Fayolle, P.J.B. King and I. Mitrani, "On the execution of programs by many processors," in *Proc. 9th International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pp. 217-228, 1983.

[6] R.L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal 45*, pp. 1563-1581, 1966.

[7] J.R. Gurd, C.C. Kirkham and I. Watson, "The Manchester prototype dataflow computer," *Communications of the ACM 28*,1, pp. 34-52, 1985.

[8] P. Heidelberger and K.S. Trivedi, "Queueing network models for parallel processing with asynchronous tasks," *IEEE Trans. on Computers C-31*,11, pp. 1099-1109, 1982.

[9] L. Kleinrock, *Queueing Systems: Volume 2, Computer Applications*. John Wiley & Sons, 1976.

[10] L. Kleinrock, "Power and deterministic rules of thumb for probabilistic problems in computer communications," in *Proc. International Conference on Communications*, pp. 43.1.1-43.1.10, 1979.

[11] D.J. Kuck, "A survey of parallel machine organization and programming," *ACM Computing Surveys 9*,1, pp. 29-59, 1977.

[12] D.J. Kuck, et al. "The effects of program restructuring, algorithm change and architecture choice on program parallelism," in *Proc. International Conference on Parallel Processing*, pp. 129-138, 1984.

[13] B. Kumar and T.A. Gonsalves, "Modelling and analysis of distributed software systems," in *Proc. 7th ACM Symposium on Operating Systems Principles*, pp. 2-8, 1979.

[14] K. C-Y. Kung, "Concurrency in parallel processing systems," UCLA CSD Report 840039, Computer Science Department, University of California, Los Angeles, 1984.

[15] M. Minsky and S. Papert, "On some associative, parallel and analog computations," in E.J. Jacks, ed., *Associative Information Technologies*, Elsevier North Holland, New York, 1971.

[16] R.R. Muntz and J.W.-N. Wong, "Asymptotic properties of closed queueing network models," in *Proc. 8th Princeton Conference on Information Sciences and Systems*, pp. 348-352, 1974.

APPENDIX

Theorem 2: Let A denote the average parallelism, $m_{\max}$ the maximum parallelism (the maximum number of processors that are simultaneously busy when an unlimited number are available), $S(n)$ the speedup with n processors, and $E(n)$ the efficiency with n processors. With processor sharing scheduling

$$S(n) \geq \min \left[A \; , \; \frac{nA}{n+A-1-\dfrac{(n-1)(A-1)}{m_{\max}-1}} \right]$$

and

$$E(n) \geq \min \left[\frac{A}{n} \; , \; \frac{A}{n+A-1-\dfrac{(n-1)(A-1)}{m_{\max}-1}} \right]$$

These bounds can be attained.

Proof: In an execution of a software system using processor sharing and n processors, subtasks are executed in the same groupings as in an execution with an unbounded number of processors. The only difference is that the length of those periods during which the number m of executing subtasks exceeds n is inflated by a factor of $\frac{m}{n}$. This observation yields the following expression for the execution time under processor sharing, where p_m denotes the proportion of time that m processors are simultaneously busy when an unlimited number are available.

$$T_n = T_\infty \left[\sum_{m=1}^{n} p_m + \sum_{m=n+1}^{m_{\max}} p_m \frac{m}{n} \right] \tag{1}$$

To establish the bound on speedup (from which the bound on efficiency directly follows), we first show that the above expression attains its maximum value when p_1 and $p_{m_{\max}}$ are the only non-zero proportions, under the constraint of a fixed average parallelism A. For suppose that $p_k>0$ for some k such that $1<k<m_{\max}$. We can reduce p_k to zero, increase p_1 by $p_k \frac{m_{\max}-k}{m_{\max}-1}$, and increase $p_{m_{\max}}$ by $p_k \frac{k-1}{m_{\max}-1}$, while keeping A fixed and ensuring that the proportions still sum to one. It is easily verified that this change does not decrease the value of equation (1): note that

$$\text{for } m_{\max}>n \geq k \; , \quad p_k \frac{m_{\max}-k}{m_{\max}-1} + p_k \frac{k-1}{m_{\max}-1} \frac{m_{\max}}{n} \geq p_k$$

$$\text{for } m_{\max} \geq k > n \; , \quad p_k \frac{m_{\max}-k}{m_{\max}-1} + p_k \frac{k-1}{m_{\max}-1} \frac{m_{\max}}{n} \geq p_k \frac{k}{n}$$

$$\text{for } n \geq m_{\max} \geq k \; , \quad p_k \frac{m_{\max}-k}{m_{\max}-1} + p_k \frac{k-1}{m_{\max}-1} \geq p_k$$

Hence, equation (1) attains its maximum value when p_1 and $p_{m_{\max}}$ are the only non-zero proportions.

From the constraints $p_1 + p_{m_{\max}} = 1$ and $p_1 + p_{m_{\max}} m_{\max} = A$, p_1 and $p_{m_{\max}}$ can be derived as $\frac{m_{\max}-A}{m_{\max}-1}$ and $\frac{A-1}{m_{\max}-1}$, respectively. Substitution in equation (1) yields, after simplification

$$T_n = T_\infty \left[1 + \frac{A-1}{n} - \frac{n-1}{n} \frac{A-1}{m_{\max}-1} \right]$$

if $n<m_{\max}$, and T_∞ otherwise. From the definition of speedup, and the fact that the execution time with only a single processor is $T_\infty A$, this establishes the desired result.

The above proof also shows that the bounds can be attained: we need only consider a software system in which some portion is sequential, and the remainder has a fixed parallelism $m_{\max}$.

QED

Corollary 2.1: Let A denote the average parallelism, f the fraction of work that is inherently sequential, and $S(n)$ the speedup with n processors. With processor sharing scheduling and $n \geq 2$

$$S(n) \geq \frac{nA}{n+A-1-(1-fA)}$$

(fA will always be between 0 and 1.)

Proof: For the processor sharing discipline, the execution time T_n is given by equation (1). Note that, in our notation, $f = \frac{T_\infty p_1}{T_\infty A} = \frac{p_1}{A}$.

Using equation (1), it is straightforward to show that T_n attains its maximum value when p_2 and $p_{m_{\max}}$ ($m_{\max}>2$) are the only other non-zero proportions, under the constraint of fixed p_1 and A. For suppose that $p_k>0$ for some k such that $2<k<m_{\max}$. We can reduce p_k to zero, increase p_2 by $p_k \frac{m_{\max}-k}{m_{\max}-2}$, and increase $p_{m_{\max}}$ by $p_k \frac{k-2}{m_{\max}-2}$, while keeping p_1 and A fixed and ensuring that the proportions still sum to one. It is easily verified that this change does not decrease the value of equation (1), hence T_n attains its maximum value when p_2 and $p_{m_{\max}}$ are the only other non-zero proportions.

From the constraints $p_1+p_2+p_{m_{\max}}=1$ and $p_1+p_2 2+p_{m_{\max}} m_{\max}=A$, $p_{m_{\max}}$ and p_2 can be derived as $\frac{(A-1)-(1-p_1)}{m_{\max}-2}$ and $\frac{(m_{\max}-1)(1-p_1)-(A-1)}{m_{\max}-2}$, respectively. Assuming that $n \geq 2$, substitution in equation (1) yields, after simplification

$$T_n = T_\infty \left[1 + \frac{(A-1)-(1-p_1)}{n} - \frac{n-2}{n} \frac{(A-1)-(1-p_1)}{m_{\max}-2} \right]$$

if $n<m_{\max}$, and T_∞ otherwise. This yields

$$S(n) \geq \frac{nA}{n+A-1-(1-p_1)}$$

which is equivalent to

$$S(n) \geq \frac{nA}{n+A-1-(1-fA)}$$

QED

Corollary 2.2: Let A denote the average parallelism, f the fraction of work that is inherently sequential, and $S(n)$ the speedup with n processors. With processor sharing scheduling

$$S(n) \leq \min\left(\frac{n}{1+f(n-1)}, A\right)$$

Proof: For simplicity we will restrict our attention to integral A. (The bound actually is unnecessarily weak for non-integral A, but the possible improvement is so minor for reasonably large A that we ignore it.) We similarly assume that $\frac{A-p_1}{1-p_1}$ is integral. It is then straightforward to show that equation (1) is minimized when p_1 and $p_{\frac{A-p_1}{1-p_1}}$ are the only non-zero proportions, under the constraint of fixed A and p_1. The resulting speedup can then be shown to be $\min\left(\frac{n}{1+\frac{p_1}{A}(n-1)}, A\right)$, or $\min\left(\frac{n}{1+f(n-1)}, A\right)$.

QED

Theorem 3: With processor sharing scheduling, an increase in the number of processors from n to kn $(n,k \geq 1)$ affects speedup as follows:

$$\max(1\,,\ \frac{kA}{(k-1)n+A}) \leq \frac{S(kn)}{S(n)} \leq \min(1+(A-1)\frac{k-1}{kn-1}\,,\ k) \tag{2}$$

Correspondingly, efficiency is affected as follows:

$$\max(\frac{1}{k}\,,\ \frac{A}{(k-1)n+A}) \leq \frac{E(kn)}{E(n)} \leq \min(\frac{1}{k}+(A-1)\frac{1-\frac{1}{k}}{kn-1}\,,\ 1)$$

These bounds can be attained.

Proof: The claims regarding the change in efficiency follow directly from those regarding the change in speedup, since efficiency (average processor utilization) is just speedup divided by the number of processors. Therefore, only the claims regarding the change in speedup will be considered.

An increase in the number of processors from n to kn increases speedup by a factor equal to

$$\frac{S(kn)}{S(n)} = \frac{\sum_{m=1}^{n} p_m + \sum_{m=n+1}^{m_{max}} p_m \frac{m}{n}}{\sum_{m=1}^{kn} p_m + \sum_{m=kn+1}^{m_{max}} p_m \frac{m}{kn}} \tag{3}$$

It is easy to show that this expression is no less than 1 and no more than k. Thus, it is only necessary to consider the second lower bound and the first upper bound in expression (2). Noting that the second lower bound is effective only for $n<A$, and that the first upper bound is effective only for $kn>A$, we need only consider these two bounds for $n<A$ and $kn>A$, respectively.

We can restrict attention to those cases in which all of the proportions p_i for $n+1 \leq i \leq kn-1$ are zero. For suppose that some p_i in this range is greater than zero. We can reduce p_i to zero, increase p_n by $p_i \frac{kn-i}{kn-n}$, and increase p_{kn} by $p_i \frac{i-n}{kn-n}$, while keeping A fixed and ensuring that the proportions still sum to one. It is easily verified that the value of expression (3) is not affected by this change.

If all p_i for $i \geq kn$ are also zero, the speedup does not change with the increase in the number of processors, and the theorem holds. Assume, therefore, that the p_i for $n+1 \leq i \leq kn-1$ are zero, but that there is at least one p_i for $i \geq kn$ that is greater than zero. The following expression is then equivalent to expression (3)

$$\frac{S(kn)}{S(n)} = \frac{\dfrac{\sum_{m=1}^{n} p_m}{\sum_{m=kn}^{m_{max}} p_m m} + \dfrac{1}{n}}{\dfrac{\sum_{m=1}^{n} p_m}{\sum_{m=kn}^{m_{max}} p_m m} + \dfrac{1}{kn}} \tag{4}$$

Expression (4) is minimized when the ratio of sums in this expression is maximized. Under the assumption that $n<A$, we apply a technical lemma, stated and proved as Lemma 3.1 below, with $l=n$ and $j=kn-n$, to show that the ratio of sums is bounded above by $\frac{1}{A-n}$. Thus, for $n<A$

$$\frac{\dfrac{1}{A-n}+\dfrac{1}{n}}{\dfrac{1}{A-n}+\dfrac{1}{kn}} \leq \frac{S(kn)}{S(n)}$$

Simplification yields the second lower bound in (2), establishing the desired lower bound on the change in speedup.

For $n \geq A$, the lower bound is attained by any software system that does not attain a parallelism of greater than n during its execution. For $n < A$, it can be easily checked that the bound is attained in the limit as $m_{\max} \rightarrow \infty$ by a software system whose execution consists of two phases; one during which its parallelism is n, and a second during which its parallelism is $m_{\max}$.

Expression (4) is maximized when the ratio of sums in the expression is minimized. Assuming that $kn > A$, Lemma 3.1 applied with $l=n$ and $j=kn-n$ shows that the ratio of sums is bounded below by $\frac{kn-A}{kn(A-1)}$. Thus

$$\frac{\frac{kn-A}{kn(A-1)} + \frac{1}{n}}{\frac{kn-A}{kn(A-1)} + \frac{1}{kn}} \geq \frac{S(kn)}{S(n)}$$

Simplification yields the first upper bound in (2), establishing the desired upper bound on the change in speedup.

For $kn \leq A$, the upper bound is attained by a software system with constant parallelism A. For $kn > A$, it can be easily checked that the bound is attained by a software system whose execution consists of two phases; one during which execution is sequential, and one during which the parallelism is equal to kn.

QED

Lemma 3.1: For any positive integers l, j such that $p_i = 0$ for $l < i < l+j$,

$$\frac{1}{A-l} \geq \frac{\sum_{m=1}^{l} p_m}{\sum_{m=l+j}^{m_{\max}} p_m m}$$

for $l < A$, and

$$\frac{\sum_{m=1}^{l} p_m}{\sum_{m=l+j}^{m_{\max}} p_m m} \geq \frac{l+j-A}{(l+j)(A-1)}$$

for $l+j > A$.

Proof: To establish the upper bound on the ratio of sums, note that, for $l < A$,

$$\frac{\sum_{m=1}^{l} p_m}{A - l \sum_{m=1}^{l} p_m} \geq \frac{\sum_{m=1}^{l} p_m}{\sum_{m=l+j}^{m_{\max}} p_m m}$$

The upper bound then results from the fact that $\sum_{m=1}^{l} p_m < 1$.

To establish the lower bound on the ratio of sums, it must be shown that the ratio of sums is minimized when p_1 and p_{l+j} are the only non-zero proportions (given that $p_i = 0$ for $l < i < l+j$). This can be shown in two steps. First, it can be shown that the value of the ratio is not increased when the proportions p_i for $1 < i \leq l$ are reduced to zero by increasing p_1 and p_{l+j} (while keeping A fixed, and ensuring that the proportions still sum to one). Second, it can be shown that the value of the ratio is not increased when the proportions p_i for $i > l+j$ are reduced to zero by correspondingly increasing p_{l+j} and decreasing p_1. (This is possible since $l+j > A$.) Solving for p_1 and p_{l+j} in terms of A yields the stated lower bound.

QED

Theorem 4: Let p_m denote the proportion of time that m processors are simultaneously busy when an unlimited number are available, and let $m_{\max}$ denote the maximum parallelism (the maximum number of processors that are simultaneously busy when an unlimited number are available). Under the processor sharing discipline, the execution time – efficiency profile for any software system has a unique knee. The number of processors that attains the knee, i.e., that maximizes $\frac{E(n)}{T_n}$, is given by

$$n = \frac{\sum_{m=\lfloor n \rfloor+1}^{m_{max}} p_m m}{\sum_{m=1}^{\lfloor n \rfloor} p_m} \tag{5}$$

if this equation has a solution; otherwise, it is given by the unique integer n satisfying

$$\frac{\sum_{m=n+1}^{m_{max}} p_m m}{\sum_{m=1}^{n} p_m} \le n \le \frac{\sum_{m=n}^{m_{max}} p_m m}{\sum_{m=1}^{n-1} p_m} \tag{6}$$

Proof: A knee of the execution time – efficiency profile occurs at a point at which the ratio of efficiency (average processor utilization) to execution time is maximized. Since $\frac{E(n)}{T_n} = \frac{S(n)}{nT_n} = \frac{T_1}{nT_n^2}$, maximizing $\frac{E(n)}{T_n}$ is equivalent to minimizing $\sqrt{n}\,T_n$. By equation (1), this is equivalent to minimizing the following function of n:

$$\sqrt{n} \sum_{m=1}^{\lfloor n \rfloor} p_m + \frac{1}{\sqrt{n}} \sum_{m=\lfloor n \rfloor+1}^{m_{max}} p_m m \tag{7}$$

Note that function (7) is a continuous function of n that tends to infinity for $n \to \infty$ and also for $n \to 0$, and that is equal to 1 for $n=1$. Hence, function (7) has a minimum, showing existence of the knee.

To show uniqueness, it is first necessary to define the derivative of function (7). At nonintegral points, the standard definition applies. At integral points, we define the derivative to be the right derivative. Note that at nonintegral points the derivative is a continuous function, but that at the integral points there may be discontinuities.

Uniqueness is then shown in two steps. First, we show that there is at most a single nonintegral point at which the derivative of function (7) is zero, and that this point, should it exist, is a minimum of function (7). Second, we show that there is at most a single integral point at which the derivative changes sign, and that this point, should it exist, is a minimum of function (7). Since it is easy to see that both points cannot exist simultaneously, these two facts are sufficient to show uniqueness.

The derivative of function (7) is given by

$$\frac{1}{2\sqrt{n}} \sum_{m=1}^{\lfloor n \rfloor} p_m - \frac{1}{2n\sqrt{n}} \sum_{m=\lfloor n \rfloor+1}^{m_{max}} p_m m$$

which is zero at each point n satisfying

$$n = \frac{\sum_{m=\lfloor n \rfloor+1}^{m_{max}} p_m m}{\sum_{m=1}^{\lfloor n \rfloor} p_m} \tag{8}$$

Since the left-hand side of this equation is a strictly increasing function of n, while the right-hand side is a nonincreasing function of n, the equation can have at most one solution. Therefore, there is at most a single point at which the derivative of function (7) is zero. It is straightforward to verify that the second derivative of function (7) is positive at this point. (If the point is an integer, the second derivative is defined as the right second derivative.) If the point is nonintegral, this implies that the point must be a minimum of function (7).

Now, suppose that there is an integer n at which the derivative of function (7) changes sign. If the sign changes from positive to negative, it must be the case that

$$\frac{1}{2\sqrt{n}} \sum_{m=1}^{n-1} p_m - \frac{1}{2n\sqrt{n}} \sum_{m=n}^{m_{max}} p_m m \ge 0$$

and

$$\frac{1}{2\sqrt{n}}\sum_{m=1}^{n} p_m - \frac{1}{2n\sqrt{n}}\sum_{m=n+1}^{m_{max}} p_m m \le 0$$

implying that

$$n \ge \frac{\sum_{m=n}^{m_{max}} p_m m}{\sum_{m=1}^{n-1} p_m}$$

and

$$n \le \frac{\sum_{m=n+1}^{m_{max}} p_m m}{\sum_{m=1}^{n} p_m}$$

These last two relations can be satisfied simultaneously only if both are equalities. In this case, however, it is easy to verify that the derivative must be positive over the interval $(n, n+1)$, in contradiction to our assumption that the derivative changes sign from positive to negative at n.

If there is an integer n at which the derivative of function (7) changes sign, the sign must therefore change from negative to positive, yielding

$$\frac{1}{2\sqrt{n}}\sum_{m=1}^{n-1} p_m - \frac{1}{2n\sqrt{n}}\sum_{m=n}^{m_{max}} p_m m \le 0$$

and

$$\frac{1}{2\sqrt{n}}\sum_{m=1}^{n} p_m - \frac{1}{2n\sqrt{n}}\sum_{m=n+1}^{m_{max}} p_m m \ge 0$$

These relations imply that

$$n \le \frac{\sum_{m=n}^{m_{max}} p_m m}{\sum_{m=1}^{n-1} p_m} \tag{9}$$

and

$$n \ge \frac{\sum_{m=n+1}^{m_{max}} p_m m}{\sum_{m=1}^{n} p_m} \tag{10}$$

From relation (10), it follows that

$$n+1 > \frac{\sum_{m=n+1}^{m_{max}} p_m m}{\sum_{m=1}^{n} p_m}$$

Since the right-hand side of relation (9) is a nonincreasing function of n, this last relation implies that no integer larger than n can satisfy relation (9). Similarly, it can be shown that no integer smaller than n can satisfy relation (10). Therefore, there must be at most one integer at which the derivative of function (7) changes sign, and this sign change must be from negative to positive, implying that the point is a minimum of function (7).

We have now shown that there exists a unique knee. The location of the knee was determined in the proof of uniqueness (equation (8) for a nonintegral knee, and relations (9) and (10) for an integral knee), and matches that given in the theorem statement. (Note that if equation (5) has an integral solution, this solution also satisfies relation (6).)

QED

Theorem 5: Under the processor sharing discipline, when the number of available processors is equal to K (achieving the knee of the execution time – efficiency profile), the attained speedup is *at least* 50% of the maximum possible, the efficiency is *at least* 50%, the utilization of the last processor is *at least* 50%, and the utilization of a single additional processor is *no more than* 50%. These bounds can be attained in the limit as $A \rightarrow \infty$.

Proof: Consider first the attained speedup. The execution time with K processors is given by

$$T_\infty \left[\sum_{m=1}^{\lfloor K \rfloor} p_m + \sum_{m=\lfloor K \rfloor+1}^{m_{\max}} p_m \frac{m}{K} \right]$$

Since, from equation (5) and relation (6), $K \geq \sum_{m=\lfloor K \rfloor+1}^{m_{\max}} p_m m$, it follows that the execution time with K processors can be at most $2T_\infty$. Therefore, the attained speedup with K processors is at least equal to 50% of the maximum possible speedup. It is straightforward to verify that this bound is attained in the limit as $A \rightarrow \infty$ by a software system whose execution consists of two phases; a first phase during which execution is sequential, and a second phase during which the parallelism is equal to A^2 (for integer A). (Note that K tends to A in this case.)

Since efficiency is the average processor utilization, if we can show that the utilization of the last processor is at least 50%, then this will also show that the efficiency is at least 50%. The utilization of the last processor is given by

$$\frac{\sum_{m=\lceil K \rceil}^{m_{\max}} p_m \frac{m}{K}}{\sum_{m=\lceil K \rceil}^{m_{\max}} p_m \frac{m}{K} + \sum_{m=1}^{\lceil K \rceil-1} p_m}$$

Since, from equation (5) and relation (6),

$$K \leq \frac{\sum_{m=\lceil K \rceil}^{m_{\max}} p_m m}{\sum_{m=1}^{\lceil K \rceil-1} p_m}$$

the utilization of the last processor is at least 50%. This bound can be attained in the limit as $A \rightarrow \infty$, since, in fact, it is possible for the efficiency as well to drop to 50% in the limit. This occurs for a software system of the same structure as that which attains the lower bound on speedup, as described earlier.

The utilization of a single additional processor is the final quantity of interest. Since $K < \lfloor K \rfloor + 1$, this utilization is bounded above by

$$\frac{\sum_{m=\lfloor K \rfloor+1}^{m_{\max}} p_m \frac{m}{K}}{\sum_{m=\lfloor K \rfloor+1}^{m_{\max}} p_m \frac{m}{K} + \sum_{m=1}^{\lfloor K \rfloor} p_m}$$

Since, from equation (5) and relation (6),

$$K \geq \frac{\sum_{m=\lfloor K \rfloor+1}^{m_{\max}} p_m m}{\sum_{m=1}^{\lfloor K \rfloor} p_m}$$

the utilization of a single additional processor is bounded above by 50%. This bound is attained in the limit as $A \to \infty$ by a software system of the same structure as that which attains the lower bound on speedup, as described earlier.

QED

Theorem 6: Under the processor sharing discipline, the number of processors K that yields the knee of the execution time – efficiency profile must satisfy

$$\frac{A}{2} \le K \le 2A-1 \tag{11}$$

When integers, these bounds on K can be attained.

Proof: Equation (5) and relation (6) yield

$$\frac{\sum_{m=\lfloor K\rfloor+1}^{m_{\max}} p_m m}{\sum_{m=1}^{\lfloor K\rfloor} p_m} \le K \le \frac{\sum_{m=\lceil K\rceil}^{m_{\max}} p_m m}{\sum_{m=1}^{\lceil K\rceil-1} p_m} \tag{12}$$

Lemma 3.1 with $l=\lfloor K\rfloor$ and $j=1$ implies, for $\lfloor K\rfloor<A$, that

$$A-\lfloor K\rfloor \le \frac{\sum_{m=\lfloor K\rfloor+1}^{m_{\max}} p_m m}{\sum_{m=1}^{\lfloor K\rfloor} p_m}$$

The same lemma with $l=\lceil K\rceil-1$ and $j=1$ implies, for $\lceil K\rceil > A$, that

$$\frac{\sum_{m=\lceil K\rceil}^{m_{\max}} p_m m}{\sum_{m=1}^{\lceil K\rceil-1} p_m} \le \frac{\lceil K\rceil(A-1)}{\lceil K\rceil-A}$$

In conjunction with relation (12), these last two relations imply that

$$A-\lfloor K\rfloor \le K$$

for $\lfloor K\rfloor < A$ and

$$K \le \frac{\lceil K\rceil(A-1)}{\lceil K\rceil-A}$$

for $\lceil K\rceil > A$, which in turn yields (with the same constraints on K),

$$A-K \le K$$

and

$$K \le \frac{K(A-1)}{K-A}$$

Simplification yields the bounds given in relation (11).

For $\frac{A}{2}$ an integer, it is straightforward to verify that the lower bound on K is attained in the limit as $m_{\max} \to \infty$ by a software system whose execution consists of two phases; one in which the parallelism is $\frac{A}{2}$, and a second in which the parallelism is $m_{\max}$. For $2A-1$ an integer, the upper bound on K is attained by a software system whose execution consists of two phases; one in which execution is sequential, and a second in which the parallelism is $2A-1$.

QED

Theorem 7: Under the processor sharing discipline, when the number of available processors is chosen to equal the average parallelism A, the attained speedup is *at least* 50% of the maximum possible, the efficiency is *at least* 50%, the utilization of the k-th last processor ($k < A$) is *at least* $\frac{k}{k+A}$, and the utilization of the k-th additional processor is *no more than* $\frac{A-1}{A+k-1}$. These bounds can be attained.

Proof: We assume that A is integral here. The claims regarding speedup and efficiency follow from Theorem 1 (III) with n substituted for by A. Only the claims regarding the utilizations of the k-th last and the k-th additional processors need be considered further.

The utilization of the k-th last processor is given by

$$\frac{\sum_{m=A-k+1}^{A-1} p_m + \sum_{m=A}^{m_{\max}} p_m \frac{m}{A}}{\sum_{m=1}^{A-1} p_m + \sum_{m=A}^{m_{\max}} p_m \frac{m}{A}}$$

It is straightforward to show that the value of this expression does not increase when p_j for $A-k<j<A$ is reduced to zero by correspondingly increasing p_{A-k} and p_A. Therefore, the utilization of the k-th last processor is bounded below by

$$\frac{\frac{1}{A}}{\frac{\sum_{m=1}^{A-k} p_m}{\sum_{m=A}^{m_{\max}} p_m m} + \frac{1}{A}}$$

Lemma 3.1 applied with $l=A-k$ and $j=k$ yields a lower bound on this latter expression of

$$\frac{\frac{1}{A}}{\frac{1}{k} + \frac{1}{A}}$$

Simplification produces the desired result.

It is straightforward to verify that the lower bound on the utilization of the k-th last processor is attained in the limit as $m_{\max} \rightarrow \infty$ by a software system whose execution consists of two phases; one in which the parallelism is $A-k$, and a second in which the parallelism is $m_{\max}$.

The utilization of the k-th additional processor is given by

$$\frac{\sum_{m=A+k}^{m_{\max}} p_m \frac{m}{A+k}}{\sum_{m=1}^{A+k-1} p_m + \sum_{m=A+k}^{m_{\max}} p_m \frac{m}{A+k}}$$

which is equal to

$$\frac{\frac{1}{A+k}}{\frac{\sum_{m=1}^{A+k-1} p_m}{\sum_{m=A+k}^{m_{\max}} p_m m} + \frac{1}{A+k}}$$

Lemma 3.1 applied with $l=A+k-1$ and $j=1$ yields an upper bound on this latter expression of

$$\frac{\frac{1}{A+k}}{\frac{k}{(A+k)(A-1)}+\frac{1}{A+k}}$$

Simplification produces the desired result.

It is straightforward to verify that the upper bound on the utilization of the k-th additional processor is attained by a software system whose execution consists of two phases; one in which execution is sequential, and a second in which the parallelism is $A+k$.

QED

The Measured Performance of Parallel Dynamic Programming Implementations

Kenneth Almquist Richard J. Anderson
Edward D. Lazowska*

Department of Computer Science
University of Washington
Seattle, WA 98195

January 9, 1989

Abstract

One focus of our overall program of research is to understand in detail the performance of algorithms on shared memory parallel machines. In this paper we study four approaches to parallel dynamic programming. By means of careful measurements, we determine how well dynamic programming can be parallelized, and we identify the bottlenecks that stand in the way of achieving the goal of linear speedup.

Keywords: parallel computing, dynamic programming, speedup

1 Introduction

One focus of our overall program of research is to understand in detail the performance of algorithms on shared memory parallel machines. By means of careful measurements, we are exploring how well certain classes of algorithms can be parallelized, and we are identifying the bottlenecks that stand in the way of achieving the goal of linear speedup.

Our approach is to take an algorithm and develop several different parallel implementations of it. We then take detailed timings of each implementation to attempt to gain a complete understanding of performance. We currently have two different parallel machines available for our experiments: a 20-processor Sequent Symmetry and a number of 5-processor

---

*Our work is supported by the National Science Foundation (Grants No. CCR-8619663, CCR-8657562, and CCR-8703049), the Naval Ocean Systems Center, U S WEST Advanced Technologies, the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program).

DEC SRC Firefly prototype workstations. Each of these systems provides hardware cache coherence by means of snooping on a shared bus.

In this paper we consider dynamic programming algorithms. Dynamic programming is an important technique with a wide range of applications. We consider the class of dynamic programming algorithms that involve the computation of the entries of a k-dimensional array where the array locations can be ordered in such a way that the value at each location is a function of locations that have already been computed. In practice it usually turns out that the values of can be computed in a very natural order.

We consider two problems that can be solved by dynamic programming and give several parallel implementations for each problem. Our implementation techniques are applicable to other dynamic programming problems of two or more dimensions. The two problems that we consider are to find the longest common substring of a pair of strings, and to compute the solution of a multiple-class queueing network performance model.

1.1 The Longest Common Substring (LCS) Problem

The longest common substring (LCS) problem is: given a pair of strings A and B, find a string C of maximum length that is a substring of both A and B. By substring, we mean a copy of a string with some of the characters removed. Formally, the string $x_1x_2\cdots x_m$ is a substring of $y_1y_2\cdots y_n$ if there exist $i_1 < i_2\cdots < i_m$ such that $x_1 = y_{i_1}, x_2 = y_{i_2}, \ldots, x_m = y_{i_m}$. The problem has many applications, including the study of the structure of proteins [Kru83].

The standard sequential algorithm [WF74] for finding the LCS of strings $a_1a_2\cdots a_n$ and $b_1b_2\cdots b_m$ constructs an $n \times m$ matrix A where $A_{i,j}$ gives the length of the LCS of the strings $a_1a_2\cdots a_i$ and $b_1b_2\cdots b_j$. The values of the matrix A can be computed by the following formula:

$$A_{i,j} = \begin{cases} \max(A_{i,j-1}, A_{i-1,j}) & \text{if } a_i \neq b_j; \\ \max(A_{i,j-1}, A_{i-1,j}, 1 + A_{i-1,j-1}) & \text{if } a_i = b_j. \end{cases}$$

It is straightforward to implement the computation of the matrix A with a pair of nested loops, yielding an $O(nm)$ algorithm. It is not difficult to recover the LCS from the matrix A.

1.2 Solving Multiple-Class Queueing Network Models (QNM)

A queueing network model (QNM) contains a set of service centers and a set of customers which travel around the network obtaining service at the various centers. If multiple customers try to obtain service at the same center simultaneously, customers are queued. The problem is to compute performance measures such as utilizations, throughputs, queue lengths, and response times, given input parameters consisting of workload intensities (the arrival rate or average population of customers) and service demands (the total service required by a customer at each resource during that customer's life in the system). The solution algorithm that we parallelized is limited to queueing networks in which the amount of service that a customer requires at a service center is exponentially distributed and in which the actions of a customer (choice of which service center to visit next, amount of service required at a service center) are independent of the actions of any other customer. Customers are grouped into classes, with all the customers in the same class having identical behavior. Customers circulate through the network forever, never entering it or leaving it.

To calculate the performance measures for customers of a particular class, it is necessary to know the queue lengths in the same network with one customer of that class removed from the network. For example, the test case used to generate most of the data for this paper had four classes of twelve customers each, which can be represented as $(12, 12, 12, 12)$. To solve this network, it is necessary to know the solution of the same network with populations $(11, 12, 12, 12)$, $(12, 11, 12, 12)$, $(12, 12, 11, 12)$, and $(12, 12, 12, 11)$.

It should be noted that there are heuristics that can rapidly produce approximate solutions to a multiple-class model. The algorithm described here, which produces an exact solution, is still of interest because no useful error bounds have been shown for the results produced by these heuristics.

The QNM problem and the LCS problem give us two very different types of grid based dynamic programming algorithms. The QNM problem naturally gives rise to higher dimensional problems since the number of dimensions is given by the number of classes. The multidimensional version of the LCS problem is not of as much interest. A bigger difference between the two problems is that the computation of each grid point for the QNM problem involves a moderate amount of work, while the computation done in the LCS problem is very simple. (For further information on queueing network models, see [LZGS84].)

1.3 Parallelizing Dynamic Programming

In order to implement dynamic programming in parallel, we must identify units of work that can be performed independently. One way to look for independent units of work is to examine the *task graph*. For concreteness, we describe the two-dimensional case, but it is straightforward to generalize the ideas to higher dimensions. If $T_{i,j}$ is the task of computing $A_{i,j}$ then $T_{i,j}$ cannot be performed until after $T_{i,j-1}$ and $T_{i-1,j}$ have been performed. This is represented in a graph by directed edges $(T_{i-1,j}, T_{i,j})$ and $(T_{i,j-1}, T_{i,j})$. We say that T' is *a predecessor of* T if there is a directed edge (T', T). A task T can be executed when all of its predecessors have been executed. A task graph algorithm is parallelized by finding subsets of the tasks that can be executed simultaneously. Figure 1 shows the task graph for dynamic programming. We define the k-th diagonal, D_k, to be the set of all tasks $T_{i,j}$ such that $i + j = k$. The key observation is that all the tasks in D_k can be executed in parallel. (There is a natural generalization to the case of greater than two dimensions.) This idea has led to a number of parallel and systolic algorithms for dynamic programming [PU84], [EW87], [LW85]. The algorithms generally solve an $n \times n$ problem with n processors in $O(n)$ time. In the next section we describe our implementations in more detail.

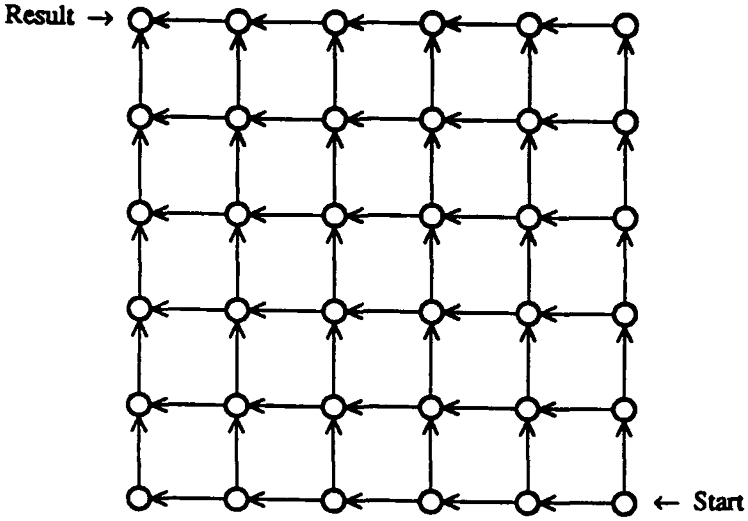


Figure 1: Task Graph

2 Implementation Strategies

In this section we motivate and sketch each of the four implementations of dynamic programming that we study in this paper. (One of the interesting aspects of our work is simply to demonstrate by implementation that there are at least four significantly different, viable approaches to implementing parallel dynamic programming.)

- **Diagonal:** The idea behind the Diagonal method is to compute all of the entries on a diagonal in a single phase. For an $n \times n$ matrix, there are $2n - 2$ phases, with the k-th phase computing all of the entries in diagonal D_k (defined above). Since all of the computations in D_k are independent, they can be done in parallel. The implementation distributes the tasks on a diagonal among the processors so that each processor computes the same number of entries (except that some processors will be short one task if the number of processors does not divide the number of tasks). When a processor completes its tasks it waits at a barrier until all the processors have finished the current phase; then the processors proceed to the next phase.
- **Pipeline:** The Pipeline method can be viewed as an attempt to directly parallelize the sequential algorithm, which computes the rows of the resulting matrix in order. If we have p processors available, we can perform the computation on the first p rows by staggering the starting times on each row. If the i-th processor starts on row i at time i, the values it needs from the preceding row will have been computed by processor $i - 1$. When a processor finishes a row, it can go on to start the next untouched row. This method turns out to be exactly the same as the Diagonal method when there are n processors.

  In attempting to get a practical implementation of this method, one important issue is to make sure that the rows remain synchronized – that the processor on row i does

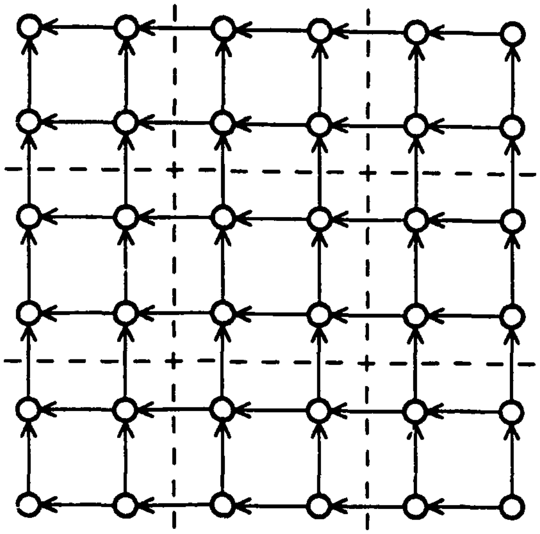


Figure 2: Partitioned Task Graph

not catch up to the processor on row $i-1$. One way to do this is to use a set of locks, so that a processor locks a column to prevent the next processor from catching up to that column. For the implementation on the LCS problem, the processors were given offsets of n/p instead of just 1. This reduced the number of locks required to keep the pipeline synchronized. The drawback was that there is a greater startup time for the pipeline to get full.

The QNM implementation did not use any explicit locks. Instead, the entries in the array containing the results were initialized to -1 so that if a processor read a value that had not yet been computed it would read a negative value – a clear signal, because computed queue lengths can never be less than zero. Before using a value, a processor would loop until the value became non-negative.

- **Task Graph:** The previous two methods use a static assignment of work to processors; the assignment is determined in advance and does not depend upon actual execution rates. A different style is to dynamically assign work to processors. One way to do this is to have a central controller which passes out pieces of work to the processors. When a processor finishes a job, it makes a call to the controller to request a new task. The controller keeps track of the precedence of the jobs, so that a job is not scheduled until after its predecessors have been completed. In our implementations, the controller was a subroutine with a single lock on a shared list data structure.

  In the QNM problem, two variations of this approach were tried, one in which the controller scheduled the tasks in first in first out (FIFO) order and one in which the controller scheduled tasks in last in first out (LIFO) order. For the LCS problem, only a FIFO order was used.

  In the LCS problem, the individual tasks are very small (they consist of a few array references, an increment, and some comparisons) so it would not make sense to schedule each task with the scheduling routine, since the cost of the scheduling would far exceed the cost of the computation. The solution is to make each task a larger unit of work. This can be done by partitioning the original set of tasks into larger groups and making each group a *supertask*. Figure 2 illustrates partitioning a 6 × 6 task graph into 9 supertasks. If the task graph for dynamic programming is partitioned into squares, it turns out that the task graph for the supertasks has roughly the same precedence structure as the original task graph. The LCS solution was implemented by partitioning the graph into supertasks, and having a scheduling routine assign supertasks to processors.

- **Synchronization-Free:** The three preceding methods, while different in many respects, are similar in that each identifies concurrently executable components of the sequential algorithm and uses some form of synchronization to coordinate the processors. Our final implementation represents an entirely different approach: we avoid all synchronization. The cost of this is that certain values may be computed multiple times. The benefit is that synchronization overhead is eliminated.

  The method is based upon a recursive definition of the problem. To solve a particular node N, the method recursively solves any unsolved predecessors, and then solves N. The key observation is that if several processors solve the same node, correctness isn't compromised.

  Three variations of this method were implemented, all for the QNM problem. They differ in how the processors select nodes for evaluation. (The objective, of course, is to minimize redundant computation without resorting to explicit synchronization.) In the first variation the processors used a pseudo-random number generator to decide which predecessor of the current node to evaluate first. (The remaining predecessors were evaluated in order following the first.) In the second variation, random choice was again used to select the initial predecessor, but nodes that were being worked on by other processors (detected by means of a flag) were skipped, and revisited in a second pass if their (recursive) solution had not yet been completed at that time. In the final approach a precomputed table, indexed by the processor number and the index of the diagonal of the graph that contains the node, was used to select the predecessor to work on first; the table was computed using a greedy heuristic which attempted to maximize the distance between the processors on a given diagonal of the graph the first time they reached that diagonal.

3 Methodology

We performed a large collection of timings on each of the implementations. We discuss our methodology in this section, summarize our results in the next section, and analyze these results in a third section.

The input data for the LCS problem involved randomly generated strings over a three-letter alphabet. The input data for the QNM problem involved several large computer system models.

On the Sequent Symmetry, timings were done using a microsecond timing facility that is built into the machine. This made it possible to get very accurate timings with virtually no overhead. On the Firefly, timings were done using a counter which the operating system increments every clock interrupt (approximately every two milliseconds). This counter was mapped into user address space, so the overhead of accessing it was low.

All of the results that we report in the tables in the following section are for the Sequent. The results on the two systems were fairly similar, and the Sequent, having more processors than the Firefly, allowed a more interesting space to be explored. Differences between the results on the two systems are noted in the text.

In reporting speedup, an "honest" figure for the speedup of a parallel implementation running with p processors is computed as the ratio of the runtime of a good sequential implementation to the runtime of the parallel implementation. Table 1 reports these "honest" speedups for three LCS implementations. In Table 2 we report the actual times (in seconds) that these implementations took with one processor, and then we give "relative" speedups in which the numerator is the runtime of the parallel implementation running with one processor rather than the runtime of a good sequential implementation. Analogously, Table 3 reports "honest" speedups for the QNM problem, while Tables 4 and 5 report "relative" speedups.

The advantage of "relative" speedups arises from the fact that our principal objective is to study the sources of performance degradation as the number of processors increases, and these are easier to spot using the "relative" measure.

4 Description of Results

Table 1 gives "honest" speedups for the dynamic programming implementations for the LCS problem. The speedups are for 1, 2, 4, 8, and 16 processors using strings of length 200 and length 1000.

One observation is that the speedup for each implementation is substantially better on the longer strings than on the shorter strings. The reason is that the units of work that each processor was given were much larger, so that the overheads in dividing the problem were amortized over more parallel work. The timings for the smaller strings give some very useful information on the bottlenecks that we encountered.

Table 1 shows that Diagonal method was the worst of the three parallel LCS implementations. Table 2 gives the actual times for each method. We can see that Diagonal method takes longer than the other methods with a single processor; we can also see that the Pipeline and Task Graph methods are comparable to the good sequential implementation when running on a single processor. Table 2 also gives relative speedups. The Diagonal

	string length 200					string length 1000				
# processors	1	2	4	8	16	1	2	4	8	16
Diagonal	0.78	1.59	2.73	4.00	4.16	0.84	1.69	3.28	5.98	9.38
Pipeline	0.96	1.70	3.07	5.00	7.04	0.97	1.80	3.47	6.72	12.18
Task	0.97	1.83	3.24	5.04	6.07	0.99	1.95	3.79	7.16	12.71

Table 1: "Honest" Speedups for the LCS Implementations

	string length 200					string length 1000				
	time	speedups				time	speedups			
# processors	1	2	4	8	16	1	2	4	8	16
Sequential	0.703					17.56				
Diagonal	0.808	1.84	3.14	4.61	4.78	20.78	2.00	3.88	7.08	11.07
Pipeline	0.730	1.77	3.19	5.20	7.30	18.13	1.86	3.59	6.94	12.59
Task	0.726	1.90	3.35	5.21	6.26	17.77	1.98	3.83	7.25	12.87

Table 2: "Relative" Speedups for the LCS Implementations

# processors	1	2	4	8	12	16
Task (FIFO)	0.80	1.56	3.04	5.88	8.30	9.08
Task (LIFO)	0.80	1.57	3.08	6.03	8.77	10.69
Pipeline	0.89	1.78	3.46	6.84	10.11	12.36
Synch.-Free	0.90	1.79	3.54	6.97	10.38	13.34

Table 3: "Honest" Speedups for the QNM Implementations

	time	speedups				
# processors	1	2	4	8	12	16
Sequential	12.88					
Task (FIFO)	16.15	1.95	3.80	7.37	10.40	11.37
Task (LIFO)	16.15	1.97	3.86	7.56	10.99	13.39
Pipeline	14.42	1.99	3.87	7.65	11.31	13.82
Synch.-Free	14.39	2.00	3.95	7.79	11.58	14.90

Table 4: "Relative" Speedups for the QNM Implementations

	time	speedups							
# processors	1	2	3	4	5	6	8	12	16
Sequential	10.10								
Pipeline	11.29	1.98	2.96	3.90	4.87	5.72	7.69	11.21	14.83
Synch.-Free	11.39	1.99	3.00	3.56	4.84	5.55	5.70	9.33	9.27

Table 5: "Relative" QNM Speedups, 200 × 200 Customers

method achieves good relative speedups when the number of processors is small compared to the size of the problem, but the speedup deteriorates rapidly as the number of processors is increased. The poor single processor performance combined with poor relative speedups when the number of processors is large accounts for the poor overall speedups shown in Table 1.

The Task Graph method is slightly superior to the Pipeline method, both in terms of the base time with one processor and in terms of the relative speedups.

For the Task Graph method, a decision needs to be made as to how many tasks are to be used. If too few tasks are used then there is insufficient parallelism, while if too many tasks are used, the overhead of the task scheduling gets too large. The figures we report are for the number of tasks that gave the best running time. For example, for strings of size 1000, the number of tasks for 2 processors was 324 and the number of tasks for 16 processors was 4096. The optimal number of tasks has not been determined, but it clearly increases with both string size and number of processors.

The speedups for the QNM problem are shown in Tables 3 and 4. All speedups are for a problem with four classes, each of which has a population of twelve. For the Pipeline and Synchronization-Free methods we have presented results for a problem with two classes, each with a population of 200, in Table 5.

We do not include results for the first two variations of the Synchronization-Free method. On the Firefly, the first variation (which randomly selected the predecessor to evaluate first) obtained a relative speedup of only 3.53 with 4 processors. For the second variation (which also used random selection but deferred processing nodes which were being worked on by other processors and came back to them later) the speedups were better, but still sufficiently poor that there appeared to be little benefit to porting the code to the Sequent. Therefore, we only implemented the final variation on the Sequent, and show the speedups for that variation here.

From the sequential timings, we see that the Task Graph method has the highest cost in the single processor case. On the Sequent, the Pipeline and the Synchronization-Free methods have approximately the same cost in the single processor case. (On the Firefly (which uses a different compiler) the Synchronization-Free method took about 10% longer than the Pipeline method, which made all the honest speedups for the Synchronization-Free method worse than those for the Pipeline method on that machine.)

On the four-class test case, the best speedups, both relative and absolute, were obtained by the Synchronization-Free method, followed by the Pipeline method, the Task Graph method using LIFO scheduling, and the Task Graph method using FIFO scheduling, in that order. This ordering applies to both the Firefly and the Sequent. The speedups for the Pipeline method are close to linear until the number of processors exceeds 13 (the length of an edge of the grid in the test case). Table 5 shows that on a test case with longer edges this effect disappears (although there is still a significant slowdown with 16 processors which we will see is due to bus loading). The Synchronization-Free method performed badly on the two-class test case with more than a few processors due to duplicate work. The curve representing duplicate work was fairly smooth except for an exceptionally high point when the number of processors was set to 4; the path selection heuristic performed particularly badly in this situation.

In all of our experiments, our parallel programs have had a specific number of processors dedicated to them. There are several reasons why this might not be possible in a general setting. First, some multiprocessor operating systems (including the one on the Firefly) don't guarantee dedicated processors to the user. We were able to obtain timings by ensuring that little else was running during our tests, but a practical implementation of an algorithm under such an operating system must be prepared to work without dedicated processors. Second, using dedicated processors requires that at least one processor be left unused by the application, so that background operating system processes will have a processor to run on. On the Firefly, with five processors, this means that 20% of the processing power of the machine will not be available to the application. Typically the operating system will use only a relatively small portion of the processor dedicated to it.

To test the effect of competition for the processors, we took a series of measurements on the Firefly using Watchtool, a performance monitor that consumes about half a CPU. The Synchronization-Free method and the Task Graph (FIFO scheduling) method were able to take advantage of the additional processing power available when allowed to compete for the fifth processor rather than being restricted to four processors, achieving relative speedups of 4.56 and 4.21, respectively. The Pipeline method, on the other hand, ran slower when allowed to compete for the fifth processor (a relative speedup of 3.17, vs. a relative speedup of 3.54 when running on four processors). The Diagonal method was not included in this test, but it would also perform badly when not run on dedicated processors because it uses barrier synchronization.

5 Analysis of Results

There are many different sources of slowdown that parallel algorithms may face. A study such as this in which a collection of implementations are examined allows us to see quite a few of these effects and to assess their impact in practice. (We note that studies of this type were pioneered by Staunstrup [MS87], who experimentally investigated a large number of "problem heap" (divide-and-conquer) algorithms.) We divide the sources of slowdown that we witnessed into seven categories:

- **Parallel Overhead:** This refers to the extra work that is done so that the algorithm can be run in parallel. For example, in some cases a different control structure is used in the parallel version of an algorithm as opposed to the sequential one. We classify anything that makes the parallel algorithm run slower on one processor as parallel overhead. All the other types of overhead we consider keep the parallel algorithm from achieving a linear speedup even after the cost of converting to the parallel algorithm has been paid.

- **Synchronization and Control Locking:** This is the cost of maintaining the parallel control. This can be divided into two components, the actual cost of the parallel primitives and the time processes spend idling while waiting for other processes. The primitives that we use for control are spin locks and barriers. Both of these entail certain costs even if no processors are required to wait at the synchronization point. On the Sequent the cost of using a lock (with no contention) is approximately the same as a procedure call and the cost of a barrier is approximately three times the cost of a procedure call. The cost of a barrier also rises with the number of processors that are used. In addition to the cost of the primitives, there are synchronization costs incurred in waiting for all processors to reach a synchronization point. In the simple case where all processors execute some code and then wait at a barrier, the cost is determined by the slowest of the processors. A processor can take longer due to spending more time executing code, due either to an imperfect division of work or because the data requires more work. A processor can also be slowed by the hardware or the operating system. For example, the processor could be interrupted to take on another process, or to service a timing interrupt.

- **Critical Sections:** A critical section is a segment of code which for some reason can only by executed by a single processor. The cost of a critical section is that processors experience delay waiting to execute the code. A critical section can be a serious bottleneck if there are a large number of processors that need to execute it.

- **Starvation:** If the algorithm does not provide sufficient parallelism then some of the processors must idle. Starvation often occurs at the start or end of a program when it is gearing up or winding down. A particularly severe case occurs when there is some inherently sequential code, so that all but one processor is idle.

- **Cache Locality:** If several processors are accessing shared data, the data may have to be moved between several caches, while in the single processor case, the data would stay within a single cache allowing faster access. It is often difficult to measure the actual amount of slowdown that this introduces.

- **Memory Contention:** A slowdown can be introduced if the rate of memory accesses by the processors is greater than the system can handle. The degree to which this is a problem varies dramatically from machine to machine. As with cache locality, the effects of this are hard to measure. We therefore measure the other causes of slowdown and assume that memory contention accounts for what we cannot measure. We would much prefer to measure memory contention directly, but we can gain some confidence in our inferences about memory contention by comparing them against our qualitative knowledge of the bus traffic generated by the program.

- **Extra Work:** There are some parallel algorithms that perform more work than their sequential counterparts. This extra work may be because some computations are done several times, or it may be because the parallel algorithm is just not as efficient as the sequential one.

For completeness, we list one other important source of slowdown of parallel programs, which was not significant in this study but which has shown up in others.

- **Data Locking:** This is the cost that is incurred to make sure that shared data is accessed correctly. It includes both the actual cost of locks and the time spent waiting for a lock to become available.

Having listed the causes of slowdown, we now discuss how they arose in each of the implementations.

5.1 The Diagonal Method

The Diagonal method was the slowest approach for the LCS problem running on a single processor. It exhibited a significant amount of parallel overhead. Nearly half of this overhead (42% for strings of length 200 and 51% for strings of length 1000) was due to stepping along the diagonals rather than along the rows. The remainder of the parallel overhead was due to the cost of scheduling.

The Diagonal method also failed to achieve linear speedup. The main problem was the cost of synchronization. The inner loop of the Diagonal method was divided between the processors. When work was begun on a given diagonal, each processor would do its share of the work and then wait at a barrier which no processor could pass until all the processors reached the barrier. The processors did not always reach the barrier at the same time. This was partially because it was not possible to give all the processors the same amount of work unless the number of tasks was a multiple of the number of processors. Furthermore, as we will see in the discussion of the Pipeline method, even if all the processors are given the same amount of work they will not necessarily complete it at the same time because the processors do not all function at exactly the same speed.

Another reason that the Diagonal method failed to achieve linear speedup is that the barrier synchronization code must be executed by each processor a fixed number of times,

# processors	2	4	8	16
12 × 12 × 12 × 12 customers	0.3%	1.7%	2.2%	11%
200 × 200 customers	0.4%	1.2%	1.9%	2.0%

Table 6: Slowdown due to Blocking in the Pipeline Method (QNM Problem)

regardless of how many processors there are. Thus the barrier operation behaves like an inherently sequential piece of code which cannot be sped up regardless of the number of processors.

5.2 The Pipeline Method

The Pipeline method was implemented for both the LCS problem and the QNM problem. It does have some overhead that was introduced in the parallelization, but less than the Diagonal method. The Pipeline method traverses the matrix in row order. However, each time a processor starts on a new row, it makes a procedure call to find out which row to use, which introduces some overhead. Additional overhead is introduced by the code to maintain synchronization.

There are four reasons why the Pipeline method did not produce linear speedup: synchronization, bus traffic, starvation, and the cost of advancing to the next row. We discuss each of these in turn.

If all the processors made progress at the same rate, processors should never block at synchronization points and the slowdown due to synchronization should be minimal. In practice, the processors do not make progress at the same rate, for reasons that include differences in clock speed, differences in cache hit ratios, and interference due to operating system activity. Table 6 shows the slowdown attributable to synchronization on the Sequent. Synchronization accounted for a bit over half of the slowdown observed for 8 or fewer processors. The slowdown for 16 processors on the first test was very large because the number of processors was greater than the length of an edge of the array (13), forcing blocking even if all the processors ran at the same speed. A second cause of slowdown was contention for the memory bus. In the Pipeline method, different processors work on adjacent rows, so that processors will always be fetching values which are in the caches of other processors, producing bus traffic. Bus traffic accounted for most of the slowdown not caused by synchronization. In the 200 × 200 QNM problem, bus contention accounted for 80% of the slowdown observed.

The Pipeline method suffers from some starvation when waiting for the pipeline to fill up initially. In the LCS problem, the processing of each row is staggered by a large amount to reduce the synchronization costs, but this increases the amount of starvation. It can be shown that starvation increases the run time by a factor of $p/2n$. If the number of processors does not divide the number of rows, there can also be starvation which can increase the run time by a factor of as much as $(p-1)/n$ in the two dimensional case.

A final cause of slowdown with more than two dimensions is the cost of selecting the next row. The program must compute the population of the first element of the row. With a larger number of processors average number of classes whose population differs between successive rows selected by a processor will increase slightly. The effect of this on speedup turns out to be negligible.

5.3 The Task Graph Method

The Task Graph method for the LCS problem performed roughly as well as the Pipeline method, but the sources of slowdown were somewhat different. The actual tasks processed were $k \times k$ submatrices, so there was little overhead associated with the approach. However, there was an overhead introduced in scheduling and manipulating the tasks. For 16 processors on strings of length 1000 the best performance was observed using 4096 tasks. The processing of the tasks slowed the method down by approximately 3%. The advantage of the Task Graph method is that the units of work can be larger, so that synchronization does not play as major a role in the slowdown. The disadvantage of the Task Graph method is that there is starvation both at the start and the end of execution. If t^2 tasks are used, then it can be shown that starvation introduces a slowdown of a factor of p^2/t^2. In the example above this is a 6% slowdown. The choice of the number of tasks to use is based on a tradeoff between the work of processing tasks and the amount of starvation. When the number of tasks is too large, the method fails miserably. This is because the task scheduling routine is a critical section. If the tasks are sufficiently large, then the critical section is not a bottleneck. However, if the tasks are so small that the time to get through the critical section is more than p times the time to execute a task, the resulting bottleneck severely restricts performance. Thus, the critical section limits the number of processors that can be used to solve the problem; at a certain point, there is no gain in applying extra processors.

The Task Graph implementation of the QNM problem did not group tasks together into supertasks, which limited the effectiveness of the method. The cost of maintaining the work list made this method slower than the others for one processor. The lock on the work list also made speedup poor. With 16 processors and FIFO scheduling, two thirds of the slowdown was due to time spend waiting for the scheduler lock. Even after adjusting the speedup figures to exclude the time waiting for the lock, we still observed a large slowdown. We believe that this is due to high bus traffic. Bus traffic should be relatively high with the Task Graph method because the assignment of tasks to processors is effectively random. Thus a processor working on a given task is unlikely to have performed the predecessors of that task, so the values computed by the tasks predecessors will have to be fetched from memory or from other processor's caches. Using supertasks would decrease the bus traffic by causing a collection of related tasks to be performed by the same processor.

Scheduling work in LIFO order rather than in FIFO order resulted in a significant performance improvement. It increased the speed of operations on the work list slightly since stack operations are slightly faster than queue operations. The LIFO approach was only 5 milliseconds faster than the FIFO approach in the single processor case, but in the 16 processor case the time spent waiting for the lock on the work list decreased by a factor of 2.5.

5.4 The Synchronization-Free Method

The Synchronization-Free method performed very well in those situations where the amount of duplicate work that was performed was held to a small proportion of the total processing. This method exhibited a certain amount of parallel overhead. It had no synchronization costs. There were still effects due to bus loading, but these were smaller than with the other methods, presumably because the amount of communication between processors was relatively small. On the Firefly we observed superlinear speedup with two to four processors. We believe that this is due to caching effects – when a processor gets a cache miss it may be able to fetch the value from the cache of another processor rather than going to memory. Duplicate work was a major problem if the processors acted randomly. On the other hand, precomputing the initial paths worked well with the 12×12×12×12 test case. In the 200×200 test case it performed well only if the number of processors was small. We conclude that the Synchronization-Free method requires that the potential parallelism be much greater than the number of processors, since otherwise the cost of duplicate work is going to be too high.

6 Conclusions

This paper has had two goals: to demonstrate through implementation that there are a number of viable parallel implementations of dynamic programming, and to understand the sources of degradation in these programs.

The Diagonal method was clearly the worst of the methods because of the parallel overhead and because of its use of barrier synchronization, which caused its performance to deteriorate rapidly as the number of processors increased and the task size became small.

The Pipeline and Task Graph methods both performed well. On the LCS problem the Task Graph method performed somewhat better when the problem size was large. We believe that this is because the Task Graph method worked with square supertasks, while the Pipeline method was limited to rectangles of width one in this implementation. On the QNM problem, where the implementation did not allow either the Task Graph method or the Pipeline method to group tasks into supertasks, the Pipeline method was clearly superior. The Task Graph method required some tuning to get the task sizes right, which makes it less attractive. For problems with more than two dimensions, the Pipeline method is less general than the Task Graph method because the Pipeline method deteriorates when given more processors than the length of a row in the matrix. The pipeline method also deteriorated badly when dedicated processors were not provided.

The Synchronization-Free method performed well when there was enough excess parallelism to hold the amount of duplicate work down. It deteriorated significantly on the two dimensional problem, which failed to supply enough parallelism to avoid lots of duplicate work with more than three processors. Our results provide a demonstration that this method outperforms approaches which use synchronization calls to avoid duplicate work when there is sufficient excess parallelism to support this approach. An area for future research is to evaluate synchronization free parallelizations of other algorithms.

In summary, three of the four methods we implemented for parallelizing dynamic programming turned out to be quite attractive. Our analysis in Section 5 provides an understanding of the four methods, and more generally of the types of bottlenecks to look for when developing other parallel algorithms.

References

[EW87] E. Edmiston and R.A. Wagner. Parallelization of the dynamic programming algorithm for comparison of sequences. In *Int. Conf. on Parallel Processing*, pages 78–80, 1987.

[Kru83] J.B. Kruskal. An overview of sequence comparison. In D. Sankoff and J.B. Kruskal, editors, *Time Warps, String Edits, and Macro-Molecules: The Theory and Practice of String Editing*, pages 1–44, 1983.

[LW85] G. Li and B.W. Wah. Systolic processing for dynamic programming problems. In *Int. Conf. on Parallel Processing*, pages 434–441, 1985.

[LZGS84] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance*. Prentice Hall, 1984.

[MS87] P. Moller-Nielsen and J. Staunstrup. Problem-heap: a paradigm for multiprocessor algorithms. *Parallel Computing*, 4:63–73, 1987.

[PU84] C.H. Papadimitriou and J.D. Ullman. A communication-time tradeoff. In *25th Symposium on Foundations of Computer Science*, pages 84-88, 1984.

[WF74] R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, January 1974.

The Use of Approximations in Production Performance Evaluation Software

John Zahorjan and Edward D. Lazowska

Department of Computer Science
University of Washington

Kenneth C. Sevcik

Computer Systems Research Institute
University of Toronto

August 1986

Abstract

Queueing network technology has become widely used in computer system performance analysis and capacity planning. The popularity of this approach is the result of the combination of speed, accuracy, and convenience that it affords.

Because separable queueing networks, which form the basis for most modelling software, do not provide a sufficiently rich set of model constructs, there has been considerable research on approximate analysis techniques for models with non-separable features, e.g., priority scheduling and memory constraints. Similarly, because even separable networks are intractable when there are many classes, efficient approximate analysis techniques have been developed for them.

In this paper we outline a number of problems that arise in using these approximate analysis techniques in software intended for production use as a capacity planning tool. There are two major lessons from our experience as the developers and maintainers of a widely-used software package: the interaction of several approximations can lead to problems not encountered when using the approximations in isolation, and the dynamics of approximations (their sensitivity to small changes in parameter values) can be more important than their average or worst case errors.

CR Categories and Subject Descriptors: C.4 [Performance of Systems]: Modeling techniques; D.2.0 [Software Engineering]: General; D.4.8 [Operating Systems]: Performance – Modeling and prediction, Operational analysis; I.6.4 [Simulation and Modeling]: Model Validation and Analysis.

General Terms: Performance

Additional Key Words and Phrases: computer system performance analysis; performance modeling software

---

This material is based upon work supported by the National Science Foundation under Grants DCR-8302383 and DCR-8352098, and by the Natural Sciences and Engineering Research Council of Canada under Grant A8654.

Authors' addresses: John Zahorjan and Edward D. Lazowska, Department of Computer Science FR-35, University of Washington, Seattle, WA 98195; Kenneth C. Sevcik, Computer Systems Research Institute, University of Toronto, Toronto, Ontario Canada M5S 1A4.

1. Introduction

Queueing network technology has been widely adopted as a capacity planning tool. Considerable work has been done on developing the techniques that have made this possible, and on examining the accuracy and efficiency of these techniques. Much of this work has been done "in the laboratory", by researchers concerned more with the abstract goal of extending the technology than with the practical use of the technology to solve real problems. At the same time, software products based on this technology, such as BEST/1 [BGS 1986] and MAP [QSP 1986], are in regular use, and considerable experience has been accumulated regarding the requirements for such tools. In this paper we report on some of these requirements, based on our observations as the developers and maintainers of the MAP software. These observations should be of value to practitioners attempting to construct useful software tools from the abstract techniques, and to researchers interested in solving problems of practical concern.

MAP has been in use on a daily basis by 50-100 users for a number of years. MAP is based entirely on approximate analysis techniques. At the lowest level is an approximate mean value analysis (MVA) solver that handles separable and some forms of non-separable models. Above this routine are a number of other routines that transform more complicated non-separable models into models of the sort solvable at the lowest level. Many of these techniques are based on material presented in [Lazowska et al. 1984].

One of the reasons for using approximate analysis techniques exclusively is flexibility. MAP models may contain a large number of separable and non-separable features. For instance, a model can include open and closed workloads, priority scheduling, memory contention, channel contention, and portions of time during which a job class is "non-dispatchable". Combinations of these features are required by our users in their roles as capacity planners. MAP imposes no "artificial" restrictions on the types of models that can be constructed from these components. For instance, any type of workload (batch, terminal, or transaction) can be subject to a memory limitation; any center can be used to represent a CPU, and there can be any number of these centers in a model; any number of centers can be priority scheduled; while there is currently a limit of 32 job classes, any class can be either open or closed independently of the others, and there is no restriction on the maximum allowable populations of the closed classes. Using approximate analysis techniques at all levels of the software aids in providing this very uniform accommodation of model features. It is important in what follows to keep in mind that there is essentially no component of MAP that calculates an exact solution. This fact has a strong influence on the techniques that can be employed in the tool, and on the types of problems encountered in constructing and using it.

The purpose of this paper is to illustrate the differences in emphasis between research in queueing network technology and the application of that technology in practice. There are two broad themes to the material:

- The interaction of approximate analysis techniques can lead to behavior not revealed by the examination of each technique in isolation.
- The most common measure of the accuracy of approximate analysis techniques, average error, is not a sufficient characterization of behavior, and in fact may be less important than other characterizations.

For the most part we limit ourselves here to identifying problems. The solutions to these problems are not obvious, and it would certainly not be possible to investigate them all in this paper. The discussion is grouped into four sections. The first considers models for which the user has an intuitive notion of what the "exact" solution must be, and so places a tight requirement on the software. The second considers the dynamics of approximations, that is, how the performance estimates change with small modifications of the input parameter values. The third explores the interactions of two or more approximate analysis techniques employed in the analysis of a single model. The fourth discusses problems of convergence, which are ubiquitous because of the widespread use of iteration in approximate analysis techniques.

2. The Need for "Exact" Solutions

There are many situations in which the user has a good idea about how the model should behave. Sometimes this means that he knows the exact solution of the model, but more often it means that he knows some relationship among the output performance measures that must hold. (For example, he might know that the response times of two classes must be equal without knowing what this response time should be.) It is important that the software produce the expected results in these cases. Even small deviations can cause the user to lose confidence in the overall approach, despite the fact that these errors may be well within what one's reasonable expectations should be about obtainable accuracy.

2.1. Priority Scheduling – The Null Process

Occasionally a user will introduce into a model a *null process*, that is, a low priority process that does nothing but consume CPU service. In the model this is usually represented as a single batch type customer with service demands of zero everywhere but at the CPU center. While the user does not know what the exact throughput of this customer should be, he does know that it should consume all CPU cycles not consumed by higher priority customers, that is, that the total CPU utilization should be 100%.

Unfortunately, techniques based on the "most accurate" priority modelling approximation [Bryant et al. 1984] do not produce this result[1]. (The "shadow CPU technique" [Sevcik 1977] does have the correct behavior, but is less reliable overall, and so is not necessarily the approach of choice.) The problem here is easily seen by examining the form of the MVA residence time equation

$$R = \frac{D\,(1+\hat{Q}_{gt})}{1-\hat{U}_{gt}}$$

where R is the residence time at the CPU of the null process, D is its service demand there, $\hat{Q}_{gt}$ is a queue length that depends on the specific priority approximation, but in general represents the arrival instant queue of equal and greater priority customers, and $\hat{U}_{gt}$ also depends on the specific approximation but represents roughly the utilization of the CPU by higher priority customers while a low priority customer is in the CPU queue. Since there is a single null process, its throughput (by Little's law) is $\frac{1}{R}$, and so its utilization is $\frac{1}{R} \times D = \frac{1-\hat{U}_{gt}}{(1+\hat{Q}_{gt})}$. This quantity should equal $1 - U_{gt}$, the idleness of the CPU with respect to the higher priority customers at equilibrium. It is only a coincidence if this actually occurs.

The solution to this problem is not clear. A brief examination of an exact expression for the response time at a priority center under the assumption that arrivals of all classes are Poisson illustrates the complexity of the problem, though. For a preemptive priority center the residence time per visit of a class j customer, R_j, is given by

$$\begin{aligned} R_j = {} & S_j \\ & + \sum_{\substack{\text{all classes } c \text{ s.t.} \\ P(c) \geq P(j)}} (\text{average arrival queue of class } c \text{ customers}) \times S_c \\ & + R_j \sum_{\substack{\text{all classes } c \text{ s.t.} \\ P(c) > P(j)}} (\text{average interrupt rate of class } c\,) \times S_c \end{aligned}$$

where $P(c)$ indicates the priority of class c and S_c is the mean service time per visit of a class c customer at the priority center. Approximations for preemptive priority essentially make convenient assumptions about the second and third terms of this expression. For instance, shadow CPU assumes that the average arrival instant queue length of all higher priority classes is zero, and that the interrupt rate is equal to the equilibrium throughput. The Bryant et al. [1984] technique assumes that the arrival instant queue lengths are the equilibrium queue lengths, and that the interrupt rate is given by the equilibrium throughputs when the population of each higher priority class is reduced by its equilibrium mean queue length at the priority center with the full population.

---

[1] Bryant et al. do not say how to build an efficient priority approximation when using an approximate analysis technique for separable models as a base. The difficulty is that performance measures for only a single population level are obtained by the base approximation, while the Bryant et al. technique requires performance measures for a potentially large number of distinct populations. Thus, an adaptation of their technique has been employed in MAP. Eager [1986] has looked at this problem independently, and has developed a different technique.

Neither of these assumptions is valid for all models, and there does not appear to be any simple set of assumptions that can give accurate answers in all cases. The difficulty in the null process model is in estimating the arrival instant queue length. The equilibrium mean queue lengths can be much too large, because the arrival of the null process is not a random event. In fact, it is clear that at the instant the null process completes a burst of service there cannot be any higher priority customers present, since the null process would not be in service otherwise. Thus, if one thinks of the null process as making many visits of a small service requirement each (this assumption is useful in formulating an approximation because the service times of all classes can be assumed to be equal), all but the initial visit to the priority center find it free of high priority customers.

In summary, in the null process case the user has a very strong intuition concerning the solution that should be provided by the software, but the problem is sufficiently complicated that it is difficult for the software to comply, and even more difficult to do so efficiently.

2.2. Symmetric Priority Distributions

MAP allows the user to specify a distribution of priority levels to be associated with each class. This distribution indicates the fraction of the total service demand at each priority center that is obtained at each possible priority level. For instance, class *A* might receive 35% of its service at the CPU at priority level 15 and the remaining 65% at priority level 4. (In general, any number of levels is allowed for each class.)

In this case, like the previous one, the user does not know what the exact solution of the model should be, but he does know that a model consisting of two identical classes and a single, priority scheduled center should give identical results for both classes. This property is not hard to guarantee, and falls out of almost any reasonably implemented approximate MVA approach. However, the user also expects that the result of the priority distribution model should be equivalent to that obtained if the priority scheduling were replaced with processor sharing (PS). This property is much more difficult to ensure.

The cause of the trouble here is similar to that in the null process case – the MVA residence time equations do not guarantee that the required property will be observed. Thus, again, it is merely a coincidence if the software obtains the expected solution.

2.3. Equivalent Open and Closed Models

Suppose the user creates a model with a single open class and obtains its solution, and then converts it to an "equivalent" closed model by changing the open class to a closed class with a very large number of customers and introducing a "request generator" consisting of a FCFS server with service time equal to the inverse of the open class arrival rate. The user expects that the solution of this model should be identical to that of the open model.

For simple, separable models this does not require any special treatment in the software. The solution obtained by the Bard-Schweitzer approximation [Bard 1979; Schweitzer 1979], for instance, will be asymptotically identical to the exact solution of the open model (which is easily computed by the software). However, let us assume that there is some complicating aspect of the model that either makes it non-separable or requires the use of more sophisticated separable model approximations than Bard-Schweitzer. For instance, suppose some center in the model is a multiprocessor with priority scheduling. This requires a solution approach such as modification of the residence time equation. The simplest way to implement this approximation is to consolidate the code for open and closed classes, that is, to consider an open class to be a closed class with an infinite population. This makes it possible to write the residence time equation for the open classes in a manner identical to that used for the closed classes. For instance, taking this approach in a separable network (for illustrative purposes) the residence time equation for the open class would be

$$R = D\,(1 + \frac{N-1}{N} Q)$$

$$= D\,(1 + Q) \qquad (1)$$

since we have assumed an infinite population ($N = \infty$). This equation is more convenient than the one usually used

$$R = \frac{D}{1-U} \qquad (2)$$

where U is the center utilization, since there is no longer any need to distinguish between open and closed classes.

This approach works fine almost all the time. Even for complicated approximations, starting with this form for the residence time equation leads to simpler code, and guarantees that equivalent open and closed models will produce the same performance measures. However, there is a catch. At high utilizations approximation (1) is unable to reach convergence in a reasonable number of steps. The problem is quite simple. Before equation (1) can be applied, we must have some estimate for the queue length Q. Using equation (1), new queue length estimates are computed as $Q \leftarrow X \times R = U(1+Q)$. Thus, for utilizations near 1.0 the queue length can grow by only about one customer per iteration. However, at utilizations near 1.0 the converged queue length is very large. Thus, the iterative scheme required by (1) fails to approach the true solution given by (2) in a reasonable number of iterations.

In summary, we are forced by numerical considerations to use modifications based on equation (2) as approximations in mixed models. This complicates the requirement to provide identical results for equivalent open and closed classes, since the closed class results must be obtained using equation (1).

2.4. Models with Memory Constraints

A common way to reflect the effect of memory is to impose a limit on the number of simultaneously memory resident customers. This feature is employed by a large number of our users, and is often necessary to obtain validated models.

The standard approach to modelling memory constraints is through decomposition [Brandwajn 1974; Lazowska & Zahorjan 1982; Brandwajn 1982]. The central subsystem is solved in isolation for each possible population of the class of interest, and the throughputs obtained are used to parameterize a flow equivalent service center. A model containing just the class of interest and the flow equivalent center is then analyzed to obtain network performance measures.

Two problems arise with this approach. The first comes about when the user examines the effect of increasing the amount of memory, and thus increases the limit on the number of memory resident customers. At some point as this limit is increased, the model should produce results identical to those obtained when the memory constraint is removed entirely. The decomposition approach will not yield this behavior in general, though, for two reasons. The first is that, for reasons of efficiency, the decomposition can be performed only approximately in multiple class networks. Thus, the decomposition solution with a large multiprogramming limit will differ from the "exact" solution obtained when the limit is removed. The second reason for the deviation is that the load dependent rates are only approximations, since an approximate analysis technique is used to solve the model. Thus, even in the single class case where an exact decomposition is feasible (for large memory constraints), the model solution will vary slightly from the exact values.

The second problem with the decomposition approach is somewhat more easily handled. It arises because the performance measures for the various classes are obtained from the solutions of separate models (one for each class) generated by the decomposition. These solutions may not represent in the aggregate a feasible solution. The most obvious example of this involves the utilization measure. The performance estimates for the total utilization of a device, obtained by summing the utilizations of the classes, may exceed 100% when decomposition is employed.

3. The Dynamics of Approximations

In assessing approximate analysis techniques for non-separable queueing models, researchers have typically used mean (and perhaps maximum) error as the measure of quality. The error measures are computed on a set of test cases selected either randomly or systematically to induce maximum stress, but in either case each test network is considered independently of the others.

In practice, the average error of a technique may not be as important as how that technique reacts to small changes in the input parameters. As mentioned previously, the normal user of this technology is greatly disturbed by any unexpected behavior, to the extent that all confidence may be lost in a tool that in fact provides a maximum error of less than 20%. The conclusion we draw is that a technique with 10% mean error that behaves consistently with intuition is preferable to a technique with 5% mean error that occasionally delivers a counter-intuitive result. It is important to note that that this is not merely a matter of psychology. Relative changes in performance measures predicted by queueing network models are often acknowledged to be more reliable than the absolute performance estimates [Lipsky & Church 1977; Lazowska et al. 1984]. Thus, a change in the "wrong" direction in response to the modification of an input parameter is quite disturbing.

3.1. Memory Modelling

Returning to the case of a model with a limit on the number of simultaneously memory resident customers, consider the implementation of the decomposition solution approach [Lazowska & Zahorjan 1982; Brandwajn 1982] in a tool based on approximate mean value analysis. Unlike exact MVA, which provides throughputs for all smaller populations in solving the model for population N, approximate MVA provides performance measures at only the full population N (or perhaps a few adjacent populations if some variant of Linearizer [Chandy & Neuse 1982] is used). Thus, to get all the required throughputs in a straightforward way requires N solutions of the model, where N is the multiprogramming limit. Because this can be quite expensive, techniques to reduce the number of solutions may be employed. Zahorjan and Lazowska [1984] suggest an interpolation scheme that requires only a constant number of solutions regardless of the multiprogramming limit. Their approach is to solve at a single "small" population and at a single "large" population and to find the parameters of a perfectly balanced network that would interpolate those two points. The throughputs of that balanced network, for which there is a simple closed form expression, are then used as approximations for the remaining throughputs.

This interpolation approach and its variants provide quite good average accuracy and speed. However, even though the average accuracy is acceptable, the dynamics of the approximation are not. In particular, if the population constraint is so large that some resource in the central subsystem is saturated, then the interpolation has a natural tendency to predict that response times will increase if the population constraint is made slightly larger. The reason for this is easily seen by examining the throughput function estimates for two different population constraints. The problem is that when the population constraint is so large that the central subsystem is saturated, the "large population" throughput used in the interpolation scheme does not increase with increasing population constraint. Thus, increasing the constraint results in an interpolation between identical throughputs, but over a larger population interval. The throughput estimates for intermediate populations are therefore slightly lower than those obtained when the interpolation is performed for a slightly smaller population constraint. Figure 1 gives an example of this effect. It shows the throughput estimates obtained by interpolating through the "exact" throughputs for populations 1 and 8 and for populations 1 and 10, under the assumption that the high population throughputs are the same.

One way to avoid this anomalous behavior is to increase the number of points through which the interpolation takes place. However, this must be done so that the interpolation points for population constraint N are a subset of those for population constraint P when $N<P$. This can lead to techniques whose cost is proportional to N, which may be prohibitively expensive for large models.

There is an additional source of poor dynamic behavior in the memory modelling approximation. Once so much memory has been added that there is no contention for it, the software should provide results identical to those obtained if the memory constraint is removed entirely. The transition from the

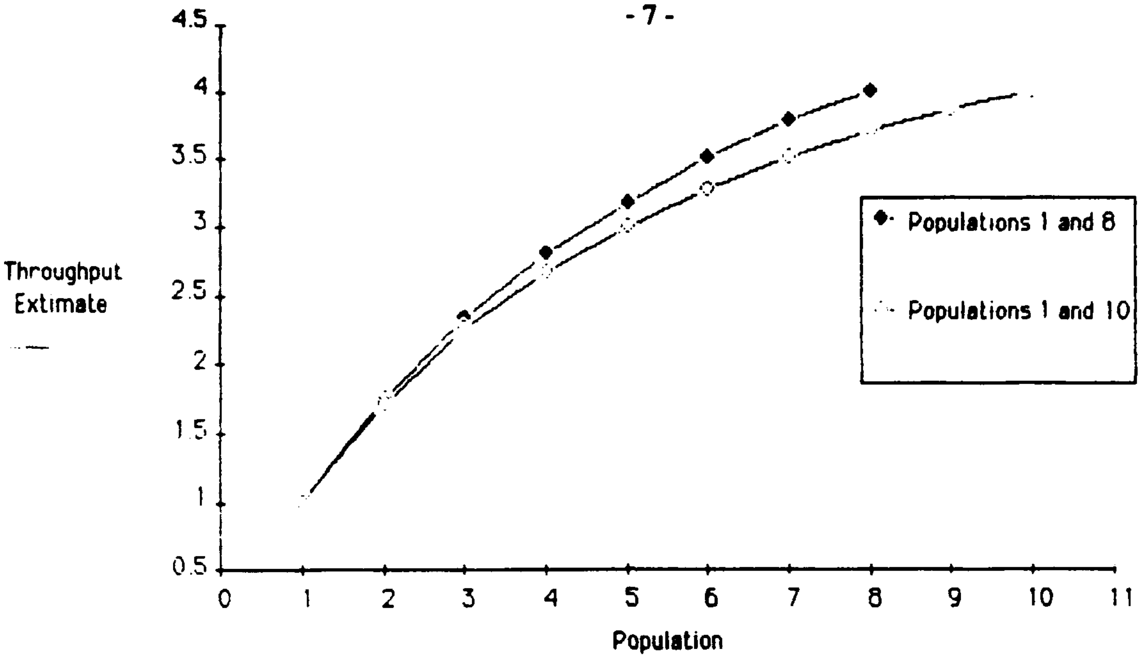


Figure 1 – Throughput Interpolation

memory modelling approximation to the solution obtained with the constraint removed causes a small discontinuity in performance measures. This discontinuity, which may be either positive or negative in sign, is typically quite small, but even a small jump with the wrong sign can be quite disturbing.

3.2. Load Dependent Servers

In a software tool based on approximate MVA, special provision must be made to accommodate load dependent service centers, such as multiprocessors. A simple technique for doing so was suggested by Neuse and Chandy [1981]. It involves a mean value assumption that represents a load dependent server by a fixed rate server with a rate equal to that experienced when the server is experiencing its mean queue length. Because the mean queue length is in general not an integer, but the load dependent service rates are specified only for integer populations, an interpolation scheme must be used.

The simplest scheme is to perform a linear interpolation between the rates at the integer populations surrounding the mean population. (This was what Chandy and Neuse recommended specifically.) This technique gives acceptable results in the cases of most interest, namely those with service rates that increase smoothly as a function of load. Thus, from an accuracy standpoint this approach should be acceptable. However, the interpolation scheme has the unfortunate property that altering the service rate for a population that is not the floor or ceiling of the average population will not cause any change at all in performance measures. While the user may only expect a very small change for "reasonable" alterations to the rate function, he is typically disturbed if no change is experienced.

In this case there are relatively simple extensions to the interpolation scheme that give the desired property. (An interpolation that uses all the rates is required.) However, these approaches require greater execution time, and can complicate efforts to provide other desirable properties in the software, and so cannot be justified on the basis of accuracy alone. Once again, it is the counter-intuitive change (or lack thereof in this case) that is important, not the average accuracy.

4. Interactions Among Approximations

A flexible software tool for capacity planning allows the representation of many system characteristics that require approximate analysis. For example, it is typical to allow mixed models (both open and closed classes) with multiprocessors, memory limitations, priority scheduling, and some form of complex I/O subsystem. While the published research provides approaches to modelling each of these characteristics, sufficient consideration has not been given to the problems encountered in combining two or more approximations. In this section we discuss a few of the difficulties we have encountered.

4.1. Approximate MVA and Memory Modelling

It is difficult in general to obtain accurate performance measures for models that are near saturation, especially for models containing open classes. When some resource is nearly fully utilized, very small errors in throughput (and thus utilization) can cause very large errors in response time. (This is easily seen by noting that the response time curve of a single queue is asymptotically vertical as the server approaches full utilization.) It is not impossible for an error of 1% in throughput to cause an error of 1000% in response time!

There also are situations in which small errors can have their effects magnified through other approximations. For example, the memory modelling approximation relies on throughput estimates conditioned on a fixed number of customers of some class being memory resident. These throughput estimates are obtained by representing the class of interest as a batch type class with the desired population and using approximate MVA to analyze the resulting model. The throughputs thus obtained will have some small error; for instance, they may all be low by a small amount. When the high level model of the memory approximation is applied, these slightly low throughputs can result in a large overestimate of response time. Even worse, if the memory is anywhere near saturated, the underestimate of throughputs causes apparent saturation to occur at lower arrival rates than it should. This can be quite noticeable, especially if the user is validating against an existing system in which he knows memory is not saturated.

4.2. Memory Modelling and Priority Scheduling

Consider once again a model with memory constraints. As noted earlier, the accepted approach to modelling this situation is through decomposition. Sauer [1981] showed experimentally that this approach gives good results.

Now consider a model in which there is priority scheduling at a heavily utilized service center, say the CPU. Suppose there are two open classes. The high priority class has an arrival rate of 1.0 and uses 75% of the CPU. The low priority class has an arrival rate of 0.1, uses 10% of the CPU, and has a memory constraint of one customer.

The standard decomposition approach for modelling memory will determine mean throughputs for all possible populations of the low priority class in the central subsystem, and will use these to parameterize a high level model consisting of the open class and a single load dependent center representing the central subsystem. In this case there is only the population of one to be considered, and since that one customer spends all its time at the CPU its response time is 4.0 (since the high priority class leaves the CPU idle only 25% of the time).

Now consider using this output rate to parameterize the high level model. That model is equivalent to a single fixed rate queue with arrival rate 0.1 and service rate 0.25. Thus, the response time prediction is 6.67. The exact solution of the priority model, however, is 26.67 (see for instance [Kleinrock 1976]).

Why did the decomposition approach fail in this case when it is generally accepted that the requirements for accurate decomposition exist in the memory modelling situation? The answer seems to be the fact that the central subsystem is priority scheduled. This causes the low priority class response times to be highly variable from visit to visit, much more so than if the CPU were scheduled using processor sharing, for instance. This high variability means that using an M/M/1 queue as the high level model is incorrect, in this case to a significant degree.

4.3. Structuring the Implementation

Aesthetically, the cleanest approach to combining approximations is layering. This is the approach advocated, for instance, by Lazowska et al. [1984]. As an example, consider a model containing a complicated I/O subsystem and a priority scheduled CPU. An I/O subsystem approximation can be used to transform the model containing the I/O path components into one containing only the CPU and disk devices, and then any of the standard priority modelling approximations can be applied to that model to represent the priority scheduling.

In practice, there are two problems with the layered approach. The first is a standard concern with any layered implementation: performance. Especially with iterative approximations, there are significant overheads associated with passing information between layers and with iterating to convergence at the lower level when the higher level is not yet near convergence.

A second problem with the layered approach is that it can very much complicate the job of developing useful approximations. For instance, consider the approximations used for priority scheduling and for multiple processors (as discussed earlier). Because both approximations are concerned with the effective instruction delivery rate of the CPU, it is in fact much more difficult to construct two independent approximations that can be used in any combination than it is to combine the approximations in one routine. In particular, ensuring that the modelling software behaves in an intuitively consistent manner is much more difficult if independent, layered approximations are employed than if a more unified approached is taken.

5. Problems with Convergence

5.1. Memory Modelling

While folklore tells us that the standard MVA based approximate analysis techniques always converge quickly (there is some formal work that demonstrates this for certain restricted models [Eager & Sevcik 1984]), we have experienced some problems caused by failure to achieve convergence. Perhaps the most important of these was encountered in the memory modelling approximation. There we must obtain throughputs of the model with the population of one class varying from 1 to some fixed limit N. Because we are working with a package based on approximate MVA, this appears to require N separate solutions, which can be quite slow.

In Section 3.1 we mentioned the use of interpolation to reduce the amount of work required. Another approach is to generate all N solutions in one call to the lowest level routine (the one implementing the MVA approximation). The potential advantage of this is that the solution of the model with $n-1$ customers can be used as the initial guess for the solution with n customers. Because the insertion of a single customer should not affect in any discontinuous way the overall behavior of model, one would expect that each successive solution would require only a few iterations, and thus that the overall cost of obtaining N solutions would be some small constant times N, as contrasted to a relatively large constant times N if using the standard initialization of distributing the customers equally among the devices.

Because this approach promises to be fast and is guaranteed to resolve the problem mentioned in Section 3 regarding counter-intuitive behavior, it was one of the first attempts we made in implementing the memory approximation. In practice, though, the time savings we had hoped to obtain were not realized. While a more intensive examination would be required to determine the cause of the inadequacy with certainty, what was clear from our experiments was that on some occasions, large errors in performance estimates were due to the failure of the model to reach a solution near convergence for the larger population values in a sequence. The reason was that the stopping criterion prematurely indicated convergence, that is, starting with a "reasonable" initial guess seemed to fool the approximation into thinking convergence had been achieved when in fact we might still be relatively far from the solution. Over the N solutions these errors would compound, so that the throughputs obtained for population N (which are often the most critical to the evaluation of the overall response time) could be substantially in error.

The apparent "fix" to this non-convergence problem is to reduce the magnitude of the stopping criterion ε, thus causing the approximation to iterate longer. However, this negates the original motivation for the approach, namely execution time efficiency. On the whole, then, some interpolation based scheme seems the most promising.

5.2. Anomalous Models

In the previous subsection we indicated that a sequence of very similar models might be difficult to solve accurately. It also is possible to encounter models for which it is difficult to compute solutions using approximate analysis techniques at all. For instance, consider a closed, single class model with a large customer population and two queueing-type service centers. The service demand at one center is a very small amount larger than that at the other. This model can be shown to have very poor convergence properties [Zahorjan et al. 1986], requiring hundreds of iterations before the estimate of the solution is within a few percent of the actual convergence point.

Unfortunately, exactly this model is sometimes encountered in practice. Suppose that a user constructs a model with a single open class, and that the bottleneck center is near saturation. Knowing that open classes tend to overestimate response times in this case, the user converts the model to an "equivalent" model with a single closed class, introducing an artificial FCFS center to control the "arrival rate" of this closed class. (The FCFS center has service demand $\frac{1}{\lambda}$, where λ was the arrival rate of the original open class.) This is exactly the case just described. Since the model was near saturation, the bottleneck center must have a service demand slightly smaller than $\frac{1}{\lambda}$, and the user is likely to introduce a closed class with a large population in substituting for the open class.

Fortunately, there are approaches that provide good behavior even for these difficult models. Some of these approaches are described by Zahorjan et al. [1986].

5.3. Nested Iterations

Suppose that the implementation contains at least two layers of approximations, both of which are iterative. There is a design question to be answered in the implementation which arises because of the interactions of the approximations, namely, should the lower level approximation iterate to convergence between steps of the higher level approximation, or should it (more nearly) take a single step for each higher level step?

While this problem has not been systematically studied in general, a careful look has been taken [Zahorjan et al. 1986] at the question applied to the levels of iterative solutions that take place in *approximate MVA solutions based on* Linearizer [Chandy & Neuse 1982]. In that case there is a "core" routine that is applied to solve the model at the full population and at each population obtained by removing one customer in turn from each closed class. We found that in roughly 25% of the cases, taking only a single step of the low level iteration for each step of the high level iteration resulted in non-convergence. In contrast, the algorithm always converged when the low level iteration proceeded to convergence between steps of the high level iteration. Thus, the latter approach seems far preferable in practice.

6. Summary

Perhaps the highest level summary of the information we have tried to convey is that building a production computer system analysis tool for use by average (rather than sophisticated) people involves much more than simply implementing a collection of the approximate analysis techniques that have appeared in the literature. There are a number of reasons for this:

- In attempting to allow the analysis of models with multiple non-separable characteristics, it is sometimes the case that there is no published technique that is directly applicable. Most approximations are formulated by introducing a single non-separable characteristic into an otherwise separable model. The task is more complicated when the remainder of the model is non-separable to start.
- The error behavior of some approximations in combination may be worse than expected. For the most part the error evaluation of approximations has been performed in isolation, that is, the only aspect of the model requiring an approximate analysis was the one for which the algorithm was specifically designed. In production tools, however, combinations of approximations are routinely employed, and understanding the causes of the errors they exhibit is more complicated.
- The mean and maximum error behavior of approximate analysis techniques may not be as important as the dynamic behavior of the techniques, that is, how they react to small changes in the input parameters. We have presented a number of instances where an approximation provided reasonably accurate results, but demonstrated incorrect dynamic behavior and so was unacceptable in practice.

- The interactions among approximate analysis techniques is important in designing the software. This in turn affects the efficiency of the tool, and the convergence properties of the approximations.

There do not seem to be any easy solutions to these problems. In developing a production performance evaluation package based on queueing network modelling technology one is confronted with the conflicting goals of accuracy and efficiency. To be useful, an appropriate balance must be struck between them. In the world of capacity planning with which we deal primarily, this balance often favors efficiency over accuracy. One must keep firmly in mind that workload characterizations and projections have very large inherent inaccuracies, and that the value of the tool lies in the analyst's ability to examine a large number of scenarios in a modest amount of time.

Despite the necessary emphasis on efficiency in construction and evaluation of the models, we have at times been attempted to buy some increased reliability and accuracy at the expense of some execution efficiency. This has not turned out to be a very profitable attitude though, for two reasons.

The first is that the kinds of problems we have experienced with the use of approximations do not seem to be overcome by better approximations, but at best the likelihood that the problems will be experienced is reduced. In this sense the problem is not "solved", it is simply made less urgent. To really overcome the problems, so that the software exhibits no irrational behavior at all, it is our guess that something approaching exact solutions are required. In part this is due to the complex interaction of the various approximations – even a small error in a specific approximation can be magnified by these interactions. Thus, a useful, complex approximation seems to require an amount of computation nearly that of the exact solution. It appears to us that anything much shy of this is more trouble than it is worth.

The second reason that using using more complicated approximations does not seem a generally useful approach is the requirement to provide a consistent modelling interface to the user, one free of artificial restrictions on the use of the available constructs. We have found that it is combinatorially harder to construct software built of many complex approximations than an equal number of relatively simple ones. This complexity impacts not only the software development effort, but also the accuracy of the resulting package. As just mentioned, the interaction of approximations is important to overall accuracy, and it is very difficult indeed to understand how many complicated techniques affect each other. This effect is so pronounced that we have in fact repeatedly discarded complex approximations after implementation in favor of simpler ones because of these problems.

Finally, it is worth pointing out that because of the simple interface provided the user with which to describe his model, the implementor has available to him significant flexibility in designing approximations. In particular, for the interface to be useful, it cannot require that the user provide a great deal of detailed parameterization. Therefore, most problems are not completely specified – some aspects of the model must be "filled in" by assumptions provided by the implementor. For instance, in providing a priority scheduling heuristic the implementor must make assumptions about the time per visit of each priority level, since the user of the software provides only the total service time requirement at the priority center. In this case, it is often convenient for the implementor to assume that every priority level has the same time per visit, but that priority levels may differ in the total number of visits made. Alternatively, it might be useful to assume that each priority level makes exactly one visit. (Although this is of course a less realistic assumption, it does have some advantages in designing an approximate analysis technique.)

Despite this modicum of leeway in deciding what the "true" solution of the model should be, there do not seem to be any easy solutions to the sorts of problems outlined in this paper. For the most part it seems that they are very difficult to address formally. For instance, realizing that the dynamic behavior of an approximation may be more important than its mean error is a matter of definition of objectives, and so would not be amenable to formal derivation. It requires experience with the use of the tool to come to this realization.

While the other problems, those involving the accuracy of approximations and the effects of their interactions, might in principle be tackled by formal techniques (either analytic or controlled experimental results), the scarcity of such results to date is an indication of the difficulty of doing so. Again, it seems that the most viable method of exploration is empirical evidence gained through the extensive use of implementations. This paper has outlined some of the problems that might be attacked using this approach.

References

[BGS 1986]
BGS Systems, Inc. *BEST/1 Reference Manual.* BGS Systems, Inc., Waltham MA, 1986.

[Bard 1979]
Y. Bard. Some Extensions to Multiclass Queueing Network Analysis. In M. Arato, A. Butrimenko and E. Gelenbe, eds., *Performance of Computer Systems*, North-Holland, 1979.

[Brandwajn 1974]
A. Brandwajn. A Model of a Time-Sharing System Solved Using Equivalence and Decomposition. *Acta Informatica 4*,1 (1974), pp. 11-47.

[Brandwajn 1982]
A. Brandwajn. Fast Approximate Solution of Multiprogramming Models. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1982, pp. 141-149.

[Bryant et al. 1984]
R.M. Bryant, A.E. Krzesinski, M.S. Lakshmi, and K.M. Chandy. The MVA Priority Approximation. *ACM Transactions on Computer Systems 2*,4 (1984), pp. 335-359.

[Chandy & Neuse 1982]
K.M. Chandy and D. Neuse. Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems. *Communications of the ACM 25*,2 (February 1982), pp. 126-133.

[Eager 1986]
D.L. Eager. The AMVA Priority Approximation. Submitted for publication.

[Eager & Sevcik 1984]
D.L. Eager and K.C. Sevcik. An Analysis of an Approximation Algorithm for Queueing Networks. *Performance Evaluation 4* (1984), pp. 275-284.

[Kleinrock 1976]
L. Kleinrock. *Queueing Systems: Volume II, Computer Applications.* John Wiley & Sons, 1976.

[Lazowska & Zahorjan 1982]
E.D. Lazowska and J. Zahorjan. Multiple Class Memory Constrained Queueing Networks. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1982, pp. 130-140.

[Lazowska et al. 1984]
E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, 1984.

[Lipsky & Church 1977]
L. Lipsky and J.D. Church. Applications of a Queueing Network Model for a Computer System. *ACM Computing Surveys 9*,3 (September 1973), pp. 205-222.

[Neuse & Chandy 1981]
D. Neuse and K.M. Chandy. A Heuristic Algorithm for Queueing Network Models of Computer Systems. Report TR-CHAN-81-02, Univ. of Texas at Austin (March 1981).

[QSP 1986]
Quantitative System Performance, Inc. *MAP Reference Manual.* Quantitative System Performance, Inc., Seattle WA, 1986.

[Sauer 1981]
C.H. Sauer. Approximate Solution of Queueing Networks with Simultaneous Resource Possession. *IBM J. Res. Develop. 25*,6 (November 1981), pp. 894-903.

[Schweitzer 1979]
P. Schweitzer. Approximate Analysis of Multiclass Closed Networks of Queues. *Proc. International Conference on Stochastic Control and Optimization*, 1979.

[Sevcik 1977]
K.C. Sevcik. Priority Scheduling Disciplines in Queueing Network Models of Computer Systems. In *Information Processing* 77, ed. B. Gilchrist, North-Holland (1977), pp. 565-570.

[Zahorjan & Lazowska 1984]
J. Zahorjan and E.D. Lazowska. Incorporating Load Dependent Servers In Approximate Mean Value Analysis. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1984.

[Zahorjan et al. 1986]
J. Zahorjan, D.L. Eager, and H. Sweillam. Accuracy, Speed, and Convergence of Approximate Mean Value Analysis. Technical Report 86-08-07, Department of Computer Science, University of Washington, August 1986.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
June 1990

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
ADVANCED NUMERICAL TECHNIQUES OF PERFORMANCE EVALUATION
VOLUME I

5. FUNDING NUMBERS
C: N66001-87-D-0136

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Washington
Department of Computer Sciences
Seattle, WA 98195

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Naval Ocean Systems Center
San Diego, CA 92152-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NOSC TD 1837

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This two volume document provides various advanced numerical techniques used in performance evaluation.

14. SUBJECT TERMS
performance evaluation

15. NUMBER OF PAGES
271

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
SAME AS REPORT

NSN 7540-01-280-5500

Standard form 298